

les Cahiers du **Programmeur**

Java EE 5

EJB 3.0 • JPA • JSP • JSF • Web Services • JMS • GlassFish • Ant

Antonio Goncalves



2^e édition

EYROLLES

les Cahiers
du **Programmeur**

Java EE 5

2^e édition

Chez le même éditeur

- S. BORDAGE. – **Conduite de projet Web.** N°12325, 5^e édition, 2008, 394 p.
- O. ANDRIEU. – **Réussir son référencement Web.** N°12264, 2008, 302 p.
- A. PATRICIO. – **Java Persistence et Hibernate.** N°12259, 2008, 364 p.
- K. DIAAFAR. – **Développement JEE 5 avec Eclipse Europa.** N°12061, 2008, 380 p.
- J.-M. DEFRANCE. – **Premières applications Web 2.0 avec Ajax et PHP.** N°12090, 2008, 450 p.
- P. ROQUES, F. VALLÉE. – **UML 2 en action.** *De l'analyse des besoins à la conception.* N°12104, 4^e édition, 2007, 382 p.
- V. MESSENGER-ROTA. – **Gestion de projet.** *Vers les méthodes agiles.* N°12165, 2007, 252 p.
- H. BERSINI, I. WELLESZ. – **L'orienté objet.** N°12084, 3^e édition, 2007, 600 p.
- L. BLOCH, C. WOLFHUGEL. – **Sécurité informatique.** *Principes et méthodes.* N°12021, 2007, 350 p.
- J. DUBOIS, J.-P. RETAILLÉ, T. TEMPLIER. – **Spring par la pratique.** *Java/J2EE, Spring, Hibernate, Struts, Ajax.* – N°11710, 2006, 518 p.
- T. ZIADÉ. – **Programmation Python.** – N°11677, 2006, 530 p.

Collection « Les Cahiers du programmeur ! »

UML 2 – Modéliser une application Web.

- P. ROQUES. – N°12136, 3^e édition, 2007, 246 p.
- Swing.** E. PUYBARET. – N°12019, 2007, 500 p.
- Java 1.4 et 5.0.** E. PUYBARET. – N°11916, 3^e édition, 2006, 400 p.
- J2EE.** J. MOLIÈRE. – N°11574, 2^e édition, 2005, 220 p.
- XUL.** J. PROTZENKO, B. PICAUD. – N°11675, 2005, 320 p.

Les Cahiers de l'Admin

- Debian Etch. Gnu/Linux.** R. HERTZOG, R. MAS. – N°12062, 2007, 428 p. avec CD-Rom.
- Sécuriser un réseau Linux.** B. BOUTHERIN, B. DELAUNAY. – N°11960, 3^e édition, 2007, 250 p.
- BSD.** E. DREYFUS. – N°11463, 2^e édition, 2004, 300 p.

Collection « Accès libre »

Pour que l'informatique soit un outil, pas un ennemi !

- Tiny ERP/Open ERP** – *Pour une gestion d'entreprise efficace et intégrée.* F. PINCKAERS, G. GARDINER. – N°12261, 2008, 276 p.

Réussir son site web avec XHTML et CSS.

- M. NEBRA. – N°12307, 2^e édition, 2008, 316 pages.

Ergonomie web. Pour des sites web efficaces.

- A. BOUCHER. – N°12158, 2007, 426 p.

Gimp 2 efficace – Dessin et retouche photo.

- C. GÉMY. – N°12152, 2^e édition, 2008, 402 p.

La 3D libre avec Blender.

- O. SARAJA. – N°12385, 3^e édition, 2008, 400 pages avec CD et cahier couleur (À paraître).

Scenari – La chaîne éditoriale libre.

- S. CROZAT. – N°12150, 2007, 200 p.

Créer son site e-commerce avec osCommerce.

- D. MERCER, adapté par S. BURRIEL. – N°11932, 2007, 460 p.

Réussir un site web d'association... avec des outils libres.

- A.-L. ET D. QUATRAVAUX. – N°12000, 2^e édition, 2007, 372 p.

OpenOffice.org 2.2 efficace.

- S. GAUTIER, C. HARDY, F. LABBE, M. PINQUIER. – N°12166, 2007, 394 p. avec CD-Rom.

PGP et GPG – Confidentialité des mails et fichiers.

- M. LUCAS, ad. par D. GARANCE, contrib. J.-M. THOMAS. N°12001, 2006, 248 p.

Ubuntu efficace.

- L. DRICOT *et al.* – N°12003, 2^e édition, 2007, 360 p. avec CD-Rom.

Réussir un projet de site Web.

- N. CHU. – N°11974, 4^e édition, 2006, 230 pages.

Collection « Poches Accès libre »

Premiers pas en CSS et HTML – Guide pour les débutants.

- F. DRAILLARD – N°12011, 2006, 232 p.

Gimp 2.4.

- D. ROBERT. – N°12295, 3^e édition, 2008, 316 p.

Firefox. Un navigateur web sûr et rapide.

- T. TRUBACZ, préface de T. NITOT. – N°11604, 2005, 250 p.

SPIP 1.9. Créer son site avec des outils libres.

- PERLINE, A.-L. QUATRAVAUX *et al.* –

- N°12002, 2^e édition 2007, 376 pages.

Mozilla Thunderbird. Le mail sûr et sans spam.

- D. GARANCE, A.-L. et D. QUATRAVAUX. – N°11609, 2005, 320 p. avec CD-Rom.

Collection « Connectez-moi ! »

Partage et publication...

Quel mode d'emploi pour ces nouveaux usages de l'Internet ?

Wikipédia. Comprendre et participer.

- S. BLONDEEL. – N°11941, 2006, 168 p.

Peer-to-peer. Comprendre et utiliser.

- F. LE FESSANT. – N°11731, 2006, 168 p.

Les podcasts. Écouter, s'abonner et créer.

- F. DUMESNIL. – N°11724, 2006, 168 p.

Créer son blog en 5 minutes.

- C. BÉCHET. – N°11730, 2006, 132 p.

Antonio Goncalves

les Cahiers
du **Programmeur**

Java EE 5

2^e édition

EYROLLES

avaxhome.ws

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Avec la contribution de Jérôme Molière



Le code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.

© Groupe Eyrolles, 2007, 2008, ISBN : 978-2-212-12363-0

À Éloïse.

Préface

Peut-être nos activités, aux uns et aux autres, nous laisseront-elles un jour le temps de regarder avec assez de recul l'aventure incroyable de cette technologie qu'est Java ? En seulement dix ans, Java s'est imposé là où on ne devinait que la domination d'un seul modèle économique. Aujourd'hui, la majorité des grands projets, tout comme la plupart des grands acteurs de l'informatique, s'appuient sur cette technologie. Pour être plus précis, il faudrait dire : « s'accrochent à sa dynamique ». Qui l'aurait parié ?

Avec l'émergence du navigateur Internet sur nos bureaux virtuels depuis le milieu des années 1990, Java est passé de ce petit bonhomme jongleur animant inutilement les pages web à cet impressionnant ensemble d'API permettant la refonte complète de nos systèmes informatiques d'entreprise.

Dans ce tourbillon technologique, nous sommes tous invités à trouver notre chemin. D'abord, les entreprises dont le métier est de s'adapter aux nouvelles contraintes et aux nouveaux modèles économiques (logiciel libre, par exemple). Mais aussi, les personnes de la galaxie informatique à qui on demande de tout savoir, sans toujours comprendre que la maîtrise de tant de concepts pose un vrai problème de compétences et de formations.

Le Conservatoire National des Arts et Métiers est l'un des interlocuteurs de ces personnes désireuses de voir leurs compétences évoluer de façon cohérente avec les offres de solutions technologiques et d'emplois. C'est dans le cadre de cette honorable et toujours jeune institution du Cnam que j'ai eu la chance de connaître Antonio Goncalves. C'est ensemble que nous nous sommes posés la question de notre contribution à ce rapport difficile entre l'évolution de la technologie et l'évolution des compétences des professionnels qui viennent nous entendre sur ces sujets.

► <http://jfod.cnam.fr>

Autant vous dire que le boulot n'est pas de tout repos ! Depuis quelques années, c'est au plus tous les deux ans que nous devons nous remettre en cause et changer non seulement de solutions mais de discours. Nos auditeurs, qui sont des professionnels, sont d'ailleurs les premiers à nous interpellier pour nous signaler que telle ou telle nouvelle solution s'impose au marché et donc aux acteurs que nous sommes. Il arrive alors que ce soit des anciens auditeurs, devenus des architectes Java EE avertis, qui, passionnés par leur métier comme par la transmission de leur savoir, viennent renforcer nos équipes pédagogiques et contribuer ainsi à la pertinence de notre offre. C'est le cas d'Antonio, qui est à la fois architecte de grands projets Java EE et enseignant au Cnam, pour la plus grande satisfaction de ses collègues et surtout de ses auditeurs.

C'est en grande partie dans ce contexte que s'inscrit le livre que vous avez entre les mains. L'idée en est née de plusieurs années de contributions à la formation Java EE au Cnam. L'orientation pragmatique de l'ouvrage est issue de la bonne connaissance de la demande de nos auditeurs.

Le pari de ce livre est de vous donner le moyen de pénétrer chacune de ces nouvelles technologies, par la pratique, et dans le cadre structurant d'un projet connu de tous pour être le projet de référence en la matière.

J'espère que ce livre aura l'audience qu'il mérite auprès de tous ceux qui ne se contentent pas de simples généralités. En tout cas, je suis sûr qu'il aura auprès de nos étudiants à Paris et dans son réseau national, l'impact pédagogique dont nous avons besoin pour relever ce défi.

Professeur Louis Dewez

Département STIC, Cnam

Avant-propos

Java EE 5

La version finale de la spécification Java EE 5 date de juillet 2006.

Retrouvez en annexe A la liste exhaustive des spécifications qui constituent Java EE 5.

Java Enterprise Edition est apparue à la fin des années 1990 et a apporté au langage Java une plate-forme logicielle robuste pour les applications d'entreprise. Remise en cause à chaque nouvelle version, mal comprise ou mal utilisée, concurrencée par les frameworks Open Source, elle a su tirer profit de ces critiques pour s'améliorer et trouver un équilibre dans sa version Java EE 5.

La deuxième édition de cet ouvrage, mise à jour et enrichie, propose de découvrir les nouveautés de cette version, tout en examinant comment les assembler pour développer un site de commerce électronique.

Objectifs de cet ouvrage

Servlet, JMS, EJB, JSP, JPA, MDB, JSF..., la liste des spécifications qui constituent Java EE 5 et qui doivent être connues par ses adeptes est longue. L'objectif de ce livre est ambitieux puisqu'il se propose de vous guider dans le développement d'un site de commerce électronique en utilisant la plupart de ces spécifications.

Java EE 5 est constitué de plus d'une vingtaine de spécifications, chacune faisant l'objet d'une description précise dans un document relativement volumineux (par exemple, 330 pages pour les servlets 2.5 ou encore 646 pour les EJB 3.0). Vous trouverez donc, dans la littérature informatique et sur Internet, une multitude de mini applications de type « Hello World » ainsi que des tutoriels couvrant chacune de ces spécifications de manière isolée. Ce n'est pas le but de cet ouvrage. Son objectif est de vous guider dans le développement d'un site complet de commerce électronique, tout en répondant à la question « Comment faire

Java Pet Store

Faisant partie du programme des BluePrints de Sun Microsystems, l'application Java Pet Store est un site de commerce électronique utilisant les spécifications Java EE.

► <http://java.sun.com/reference/blueprints/>

GlassFish

GlassFish est un serveur d'applications que Sun a donné à la communauté Open Source.

Derby

Derby est une base de données relationnelle Open Source.

Sources

Le code source de l'application développée dans ce livre est disponible en ligne sur le site :

► <http://www.antoniogoncalves.org>

Vous y trouverez aussi d'autres ressources telles qu'un forum pour déposer vos remarques ou échanger de l'information.

pour assembler ces spécifications ? ». La structure de l'application suit les règles de l'art en matière d'architecture : découpage en couches, couplage lâche et design patterns.

Afin de vous raccrocher à des concepts et pratiques connus de la communauté Java, cet ouvrage s'inspire du Java Pet Store de Sun et vous servira de guide dans le développement d'un site web proche de cette application. De plus, ce livre couvre une large partie des spécifications Java EE 5, utilise la version 5 du JDK, les design patterns, ainsi que le serveur GlassFish et la base de données Derby pour exécuter l'application. Il est abondamment illustré de diagrammes UML, d'extraits de code et de captures d'écrans. Enfin, les pages de ce livre sont accompagnées de notes, de remarques et de références pour vous permettre d'approfondir vos connaissances. Le développement de cette application est fait de manière incrémentielle, afin d'appréhender au fur et à mesure chacune des spécifications.

À qui s'adresse cet ouvrage ?

Le but de ce livre n'est pas de détailler la syntaxe du langage Java ou encore l'ensemble des méthodes des classes constituant l'API EJB. Si tel était le cas, vous ne pourriez l'emporter avec vous, à moins de posséder une brouette, en raison de son volume et de son poids.

Cet ouvrage s'adresse avant tout à des lecteurs ayant un niveau avancé en Java/UML et quelques connaissances en développement web.

Il est également dédié aux architectes souhaitant comprendre comment imbriquer les différentes API de Java EE 5 pour réaliser une application Internet-intranet.

Les débutants et les étudiants y trouveront aussi leur compte en utilisant les multiples références que contient ce livre. Ces dernières leur permettront d'approfondir un sujet en particulier.

www.avaxhome.ws

Structure du livre

Le **chapitre 1** présente l'étude de cas d'une application de commerce électronique inspirée du blueprint Java Pet Store de Sun. La société fictive YAPS veut informatiser son activité de vente d'animaux domestiques. Pour ce faire, elle a besoin d'un site pour les internautes, d'un client riche pour ses employés et de dialoguer avec ses partenaires externes (banque et transporteur).

UML, cas d'utilisation.

Le **chapitre 2** se concentre sur l'architecture technique et logicielle de l'application YAPS Pet Store. Ce chapitre présente brièvement les outils et API utilisés pour le développement.

Java 5, HTML, XML, Java EE 5, Blueprint, design pattern, UML.

L'installation et la configuration des outils se fait au **chapitre 3**.

JDK, Ant, GlassFish, Derby, TopLink.

Le développement de l'application utilisant une approche bottom-up, le **chapitre 4** entre dans le vif du sujet en développant les objets persistants.

JPA, entity.

Le **chapitre 5** rajoute une couche de traitements métiers venant manipuler les objets persistants.

EJB stateless, entity manager, JPQL.

Le **chapitre 6** nous explique comment compiler et déployer l'application pour qu'elle soit utilisée par une IHM Swing.

Ant, JNDI, Swing, GlassFish, TopLink, Derby.

Le **chapitre 7** crée une première version de l'application web qui permet de visualiser le catalogue des articles de la société et de gérer l'accès des clients.

JSP, JSTL, JSF, Unified Expression Language.

Le **chapitre 8** rajoute un panier électronique au site pour pouvoir acheter des animaux domestiques en ligne.

EJB stateful.

Le **chapitre 9** s'intéresse aux échanges B2B entre la société YAPS et ses partenaires externes (banque et transporteur).

Web Service, WSDL, Soap, JAXB.

Les traitements asynchrones, comme l'impression d'un bon de commande ou l'envoi d'e-mails, sont développés au **chapitre 10**.

JMS, message-driven bean, JavaMail.

L'**annexe A** répertorie de manière exhaustive les spécifications constituant Java EE 5. L'**annexe B** fournit le code complet des tâches ant employées pour compiler et déployer l'application. L'**annexe C** liste les sigles et acronymes que vous retrouverez dans cet ouvrage alors que l'**annexe D** compare les EJB 2.1 et les EJB 3.0. Enfin, l'**annexe E** vous guide pas à pas dans le développement d'une application web Java EE 5 utilisant l'IDE IntelliJ IDEA.

/// Top-down et bottom-up

L'approche top-down préconise de développer une application de haut en bas, c'est à-dire en commençant par les couches de présentations jusqu'aux couches d'accès aux données. L'approche bottom-up consiste à faire le chemin inverse.

Remerciements

La rédaction de ce livre n'aurait pas été possible sans l'aide et les conseils de Jean-Louis Dewez. Je tiens à le remercier pour son écoute et les multiples discussions constructives que nous avons eues.

Le graphisme de l'application web est l'oeuvre de David Dewalle, qui a aussi développé la partie Swing. Un grand merci à Alexis Midon pour m'avoir aidé dans les développements côté serveur.

Je remercie également mon équipe de relecteurs Zouheir Cadi, Alexis Midon et Matthieu Riou, pour m'avoir permis d'améliorer la qualité de ce livre grâce à leur expertise et leurs critiques.

Je remercie Egor Malyshev d'avoir participé à la rédaction de l'annexe E « Développement avec IntelliJ IDEA ».

Merci à l'équipe des éditions Eyrolles, Muriel et Karine pour leur patience et leurs encouragements, Aurélie, Sophie, Matthieu et Gaël pour le sprint final de relecture et de mise en pages.

Merci à la communauté Java et plus particulièrement à la communauté GlassFish qui m'a été d'un très grand secours. Je tiens aussi à remercier les éditeurs JetBrains (IntelliJ IDEA) et Visual Paradigm International pour m'avoir offert des licences de leurs excellents logiciels.

Un grand merci à tous ceux qui m'ont épaulé durant cette épopée (ma femme Denise et mes proches).

Table des matières

1. PRÉSENTATION DE L'ÉTUDE DE CAS 1

- Expression des besoins • 2
- Diagramme de cas d'utilisation • 3
- Les acteurs du système • 3
- Les cas d'utilisation • 4
 - Gérer les clients • 5
 - Gérer le catalogue • 7
 - Visualiser les articles du catalogue • 8
 - Diagramme d'activités • 9
 - Rechercher un article • 11
 - Se créer un compte • 12
 - Se connecter et se déconnecter • 14
 - Consulter et modifier son compte • 16
 - Acheter des articles • 17
 - Créer un bon de commande • 22
 - Visualiser et supprimer les commandes • 22
- En résumé • 23

2. ARCHITECTURE DE L'APPLICATION 25

- Présentation des langages utilisés • 26
 - Java SE 5 • 26
 - Autoboxing • 26
 - Annotations • 27
 - Génériques • 28
 - Les types énumérés • 28
 - Swing • 28
 - JNDI 1.5 • 29
 - JDBC 3.0 • 29
 - XML et XSD • 30
 - HTML et XHTML • 30
- La plate-forme Java EE 5 • 31
 - JPA 1.0 • 32
 - JMS 1.1 • 32
 - EJB 3.0 • 33
 - EJB stateless • 33
 - EJB stateful • 34
 - Message-driven bean • 34
 - Entity • 35

- Le conteneur d'EJB • 35
- Servlet 2.5 et JSP 2.1 • 36
- Langage d'expression • 37
- JSTL 1.2 • 37
- JSF 1.2 • 38
- Le conteneur de servlet • 38
- JavaMail 1.4 • 38
- JAXB 2.0 • 38
- Services web • 39

Blueprints • 39

- Java Pet Store • 40

Les design patterns • 41

UML 2 • 41

Architecture de l'application • 42

- L'architecture en trois couches • 42

Architecture applicative • 43

- Couche de présentation • 43
- Couche de navigation • 43
- Couche de traitement métier • 44
- Couche de mapping objet/relationnel • 44
- Couche de persistance • 44
- Couche d'interopérabilité • 44

Architecture technique • 44

En résumé • 45

3. OUTILS ET INSTALLATION 47

Outils utilisés pour le développement de l'application • 48

- JDK • 48
- Ant • 48
- GlassFish • 48
- Derby • 49
- Environnement de développement • 49
- Outil de modélisation UML • 49

Installation des outils • 50

- JDK 1.5 • 50
- Ant 1.7 • 52
- GlassFish V2 • 52

Configuration du serveur GlassFish • 55

L'utilitaire asadmin • 55	
Création d'un domaine • 56	
Démarrage du domaine • 57	
Configuration de la base de données • 58	
Création d'un pool de connexions • 58	
Création de la base de données • 59	
Création d'une source de données • 59	
Création des ressources JMS • 60	
Création de loggers • 61	
Récapitulatif des éléments de configuration • 63	
Environnement de développement • 63	
Les répertoires • 64	
En résumé • 65	

4. OBJETS PERSISTANTS..... 67

La persistance des données • 68	
La sérialisation • 68	
JDBC • 68	
Mapping objet-relationnel • 69	
Java Persistence API • 69	
Entity • 70	
Exemple d'entity • 70	
Annotations élémentaires du mapping • 71	
Table • 71	
Clé primaire • 73	
Colonne • 74	
Annotations avancées • 76	
Date et heure • 76	
Données non persistées • 77	
Englober deux objets dans une seule table • 78	
Relations • 79	
Jointures • 79	
Relation unidirectionnelle 1:1 • 80	
Relation unidirectionnelle 0:1 • 82	
Relation bidirectionnelle 1:n • 82	
Relation unidirectionnelle 1:n • 85	
Chargement d'une association • 87	
Ordonner une association multiple • 88	
Cascade • 89	
Héritage • 89	
Le cycle de vie d'un entity • 91	
Les annotations de callback • 92	
Les entities de YAPS Pet Store • 93	
Le catalogue • 93	
Catégorie • 94	
Produit • 95	

Article • 96	
Le client • 97	
Client • 98	
Adresse • 99	
Le bon de commande • 100	
Bon de commande • 101	
Ligne de commande • 102	
Carte de crédit • 103	
Paquetages des entités • 103	
Schéma de la base de données • 104	
En résumé • 104	

5. TRAITEMENTS MÉTIER 107

Stateless session bean • 108	
Exemple de stateless bean • 109	
Comment développer un stateless bean • 110	
Les interfaces • 110	
Interface distante • 111	
Interface locale • 112	
La classe de l'EJB • 113	
Entity manager • 114	
Contexte de persistance • 115	
Manipuler les entités • 116	
Persister un entity • 117	
Rechercher un entity par son identifiant • 118	
Rattacher un entity • 118	
Mettre à jour un entity • 119	
Supprimer un entity • 120	
Langage de requêtes • 120	
JPQL • 121	
Effectuer des requêtes en JPQL • 121	
Démarcation de transactions • 123	
Transactions • 124	
Gestion des transactions par le conteneur • 124	
Gestion des exceptions • 126	
Exceptions d'application • 126	
Exception système • 128	
Le cycle de vie d'un stateless bean • 129	
Les annotations de callback • 129	
Les stateless beans de YAPS Pet Store • 130	
La gestion des clients • 131	
CustomerLocal • 131	
CustomerRemote • 132	
CustomerBean • 132	
La gestion du catalogue • 134	
CatalogBean • 135	

La gestion des bons de commande • 135	La navigation entre pages • 184
Paquetages des stateless beans • 137	Navigation statique • 185
Architecture • 137	Navigation dynamique • 186
En résumé • 138	Comment développer une application web avec JSF • 187
6. EXÉCUTION DE L'APPLICATION 141	L'application web YAPS Pet Store • 190
Swing • 142	Décorateurs • 190
Exemple d'appel à un EJB dans Swing • 142	La visualisation du catalogue • 192
JNDI • 143	Le managed bean CatalogController • 192
Comment développer l'application Swing • 145	Les pages web • 194
Service Locator • 145	La navigation • 194
Business Delegate • 147	La page d'affichage des produits • 196
Appel d'un EJB stateless dans cette architecture • 148	La page d'affichage des articles • 197
L'application graphique YAPS Pet Store • 150	La page de détail de l'article • 198
La gestion des clients • 151	La gestion du compte par les clients • 199
La gestion du catalogue • 152	Le managed bean AccountController • 199
La gestion des bons de commande • 152	Les pages web • 200
Paquetages du client Swing • 154	La navigation • 201
Architecture • 154	L'en-tête • 203
Exécuter l'application • 154	La page de login • 203
Compiler • 155	Le formulaire de saisie • 205
Packager • 155	L'affichage du compte client • 206
Interface graphique • 156	La mise à jour du compte client • 208
Application serveur • 156	Gestion des erreurs • 209
Déployer • 156	Paquetages et répertoires de l'interface web • 211
Exécuter • 159	Architecture • 212
En résumé • 160	Exécuter l'application • 212
7. INTERFACE WEB 163	Packager • 213
Le duo Servlet-JSP • 164	Déployer l'application et accéder au site • 213
Les servlets • 164	En résumé • 214
Les JSP • 166	8. GESTION DU PANIER ÉLECTRONIQUE.....217
Le design pattern MVC • 167	Stateful session bean • 218
Le langage d'expression • 170	Exemple de stateful bean • 219
JSTL • 170	Comment développer un stateful bean • 220
JSF • 172	Les interfaces • 221
Les balises JSF • 173	La classe de l'EJB • 221
Les balises HTML • 174	Le cycle de vie d'un stateful bean • 222
Les balises Core • 176	Les annotations de callback • 222
Exemple de page JSP utilisant les balises JSF • 177	La gestion du Caddie de YAPS Pet Store • 223
Le langage d'expression unifié • 179	Le stateful bean • 223
Traitements et navigation • 180	ShoppingCartLocal • 224
La FacesServlet • 181	ShoppingCartBean • 224
Le managed bean • 182	CartItem • 226
L'injection • 183	Paquetages du stateful bean • 226
La glue entre le managed bean et la page • 184	Le managed bean • 226
	Les pages web • 228

La navigation • 228	
Ajouter un article au Caddie • 230	
La page de contenu du Caddie • 231	
La page de saisie des données de livraison et de paiement • 232	
La page récapitulative • 235	
Architecture • 236	
Exécuter l'application • 236	
En résumé • 237	
9. ÉCHANGES B2B.....	239
Les standards autour des services web • 240	
Soap • 240	
UDDI • 241	
WSDL • 241	
JAX-WS 2.0 • 242	
JAXB 2.0 • 243	
Services web • 245	
Exemple de service web • 245	
Exemple d'appel à un service web • 246	
Annotations JAX-WS • 246	
Le service • 247	
La méthode • 247	
Les paramètres de la méthode • 248	
Comment développer un service web • 250	
Développer la classe du service web • 250	
Générer les artefacts serveurs • 251	
Générer les artefacts clients • 252	
Appeler un service web • 252	
La vision globale • 254	
Les services web utilisés par YAPS Pet Store • 255	
La validation des cartes de crédit • 255	
Avertir le transporteur • 256	
Appel des services web • 257	
Paquetages des différents services web • 260	
Architecture • 260	
Exécuter l'application • 260	
Compiler • 261	
Packager • 261	
Déployer • 262	
Tester les services web avec GlassFish • 262	
Exécuter • 263	
En résumé • 265	
10. TRAITEMENTS ASYNCHRONES.....	267
JMS • 268	
Les messages • 269	
L'en-tête du message • 269	
Les propriétés • 270	
Le corps du message • 270	
Les objets administrés • 271	
La fabrique de connexions • 271	
Destinations • 272	
Le mode point à point • 273	
Le mode publication/abonnement • 273	
Envoyer les messages • 274	
Recevoir un message • 275	
La sélection de messages • 277	
Message-driven bean • 278	
Exemple de message-driven bean • 278	
Le cycle de vie d'un MDB • 280	
Les annotations de callback • 281	
JavaMail • 281	
La classe Session • 282	
La classe Message • 282	
La classe InetAddress • 283	
La classe Transport • 283	
Les traitements asynchrones de YAPS Pet Store • 284	
L'envoi du message • 285	
Les message-driven beans • 286	
Envoi d'e-mails • 286	
Impression du bon de commande • 288	
Listener JMS de l'application Swing • 289	
Paquetages des MDB • 292	
Architecture • 292	
Exécuter l'application • 293	
En résumé • 293	
A. SPÉCIFICATIONS JAVA EE 5	295
B. TÂCHES ANT	297
Build.xml • 297	
Admin.xml • 309	
C. SIGLES ET ACRONYMES	317
D. EJB 2	321
Un exemple d'entity bean • 321	
Un exemple de stateless bean • 326	
En résumé • 329	

E. DÉVELOPPEMENT AVEC INTELLIJ IDEA..... 331

Un projet façon IntelliJ IDEA • 332

Créer et configurer le projet • 332

Créer les éléments de l'application • 334

Créer l'entity Address • 335

Méthodes de callback • 337

Diagramme d'entités-relations de JPA • 338

Créer l'interface JSF • 339

Déployer et exécuter l'application • 342

En résumé • 345

INDEX347

chapitre 1



Présentation de l'étude de cas

Ce chapitre présente de manière globale l'étude de cas que nous allons développer tout au long de cet ouvrage : un site de commerce électronique, spécialisé dans la vente d'animaux domestiques. Afin de décrire les besoins de la société YAPS, nous utiliserons des diagrammes de cas d'utilisation et d'activité UML ainsi que des maquettes d'écrans.

SOMMAIRE

- Présentation de la société YAPS
- Application YAPS Pet Store
- Acheter des animaux en ligne
- Site de commerce électronique
- Expression des besoins
- Cas d'utilisation et acteurs du système

MOTS-CLÉS

- UML
- Cas d'utilisation
- Acteurs du système
- Diagramme d'activité
- Maquettes d'écrans
- Java Pet Store

avaxhome.ws

TÉLÉCHARGER YAPS Pet Store

Retrouvez le site YAPS Pet Store à l'adresse suivante :

► <http://www.antoniogoncalves.org>

UML Les créateurs du langage

James Rumbaugh, Grady Booch et Ivar Jacobs sont les créateurs du langage UML.

Cet ouvrage repose sur l'analyse du système d'information et plus particulièrement du système informatique de l'entreprise fictive YAPS. Cette société américaine vend des animaux de compagnie. Elle continue d'exercer son métier tel qu'elle le faisait à ses débuts, c'est-à-dire qu'elle répertorie ses clients et ses articles sur des fiches de papier bristol, reçoit les commandes par fax, les chèques par courrier puis envoie le bon de commande au client. Une fois le chèque encaissé par la banque BarkBank, elle utilise la société de transport PetEx pour acheminer les animaux vers leurs nouveaux propriétaires. YAPS est depuis toujours implantée dans le sud de la Californie où sont domiciliés ses principaux clients.

Récemment, elle a ouvert son marché à d'autres états américains, ainsi qu'à l'étranger. YAPS n'arrive plus à gérer manuellement cette expansion et souhaite créer un système informatique pour lui permettre de faire face à sa récente croissance. Elle attend de celui-ci qu'il lui permette de vendre ses animaux en ligne, de gérer son catalogue d'articles et sa base de données de clients. De plus, ses partenaires (la banque BarkBank et la société de transport PetEx) souhaitent avoir la possibilité d'échanger des données aux formats électroniques via Internet.

Ce système informatique est baptisé « YAPS Pet Store ». Il doit répondre à certains besoins en termes de performance et de robustesse comme la haute disponibilité puisque le site doit être accessible 24h/24 et 7j/7, et supporter un nombre élevé d'internautes. En effet, bien que YAPS soit présente dans le monde entier, la majeure partie de ses clients se trouve aux États-Unis. Il faut donc prévoir une hausse des accès au système durant la journée.

Expression des besoins

Pour exprimer les besoins de la société YAPS, nous allons utiliser le formalisme UML des cas d'utilisation. Ces derniers ont été développés par Ivar Jacobson bien avant l'apparition d'UML (*Unified Modeling Language*). Ils ont été intégrés à ce langage de modélisation pour représenter les fonctionnalités du système du point de vue utilisateur. Ils permettent de modéliser des processus métier en les découpant en scénarii. Les cas d'utilisation sont normalement représentés par un schéma, puis enrichis par un document décrivant plus précisément chaque cas ainsi que d'une maquette de l'interface graphique et/ou d'un diagramme d'activités.

Le diagramme de cas d'utilisation se compose :

- d'**acteurs** : ce sont les entités externes (personne humaine ou robot) qui utilisent le système ;
- de **cas d'utilisation** : ce sont les fonctionnalités proposées par le système.

Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation ci-après décrit les besoins de la société YAPS de façon synthétique et peut être lu comme ceci : « Un employé peut gérer les articles du catalogue, gérer les clients, visualiser et supprimer les commandes. Un internaute peut se créer un compte, visualiser et rechercher un article dans le catalogue... ».



UML Le système à étudier

Dans un diagramme de cas d'utilisation, les rectangles autour desquels gravitent les acteurs symbolisent les systèmes à étudier. Les acteurs sont représentés par une icône (appelée *stick man*), alors que les cas d'utilisation sont représentés par une forme ovale.

UML La relation <<Extend>>

Notez la présence de la relation <<Extend>>. Cela signifie que le cas d'utilisation incorpore de manière facultative un autre cas d'utilisation. Dans notre exemple, on crée un bon de commande si l'achat d'articles a été effectué.

UML La relation <<Include>>

La relation <<Include>> signifie que le cas d'utilisation incorpore explicitement et de manière obligatoire un autre cas d'utilisation. Dans notre exemple, lorsqu'on achète des articles, les données de la carte bancaire sont obligatoirement validées.

Figure 1–1
Diagramme de cas d'utilisation

Les acteurs du système

Les acteurs humains qui utilisent le système YAPS Pet Store sont les suivants :

- **Employé** : les employés de la société YAPS s'occupent de mettre à jour le catalogue des articles ainsi que la liste des clients. Ils peuvent aussi consulter les commandes passées en ligne par les clients.
- **Internaute** : il s'agit d'une personne anonyme qui visite le site pour consulter le catalogue d'animaux domestiques. Si l'internaute veut acheter un animal, il doit d'abord créer un compte. Il devient alors un client de la société YAPS.
- **Client** : un client peut visualiser le catalogue, modifier ses coordonnées et acheter des articles en ligne. Dans ce dernier cas, il reçoit une confirmation de sa commande et se fait livrer à domicile.

Il faut également mentionner ici les systèmes informatiques externes, utilisés par la société YAPS :

- **BarkBank** : YAPS délègue la validation des cartes bancaires à la banque BarBank.
- **PetEx** : la livraison des animaux est assurée par la société de transport PetEx. Celle-ci se rend à l'entrepôt de YAPS, charge les animaux dans ses camions, puis les achemine chez les clients.

Les cas d'utilisation

Chaque cas d'utilisation représenté dans le diagramme précédent doit être complété d'un texte explicatif. Bien que le formalisme de ce texte ne soit pas spécifié dans UML, il possède fréquemment les rubriques suivantes :

- **Nom** : le plus souvent le nom contient un verbe à l'infinitif puisqu'un cas d'utilisation décrit une interaction entre un acteur et le système.
- **Résumé** : une brève description du cas d'utilisation.
- **Acteurs** : cette rubrique décrit la liste des acteurs interagissant avec le cas d'utilisation.
- **Pré-conditions** (optionnel) : ce sont les conditions nécessaires pour déclencher le cas d'utilisation.
- **Description** : cette rubrique contient un texte explicitant le cas d'utilisation.
- **Post-conditions** (optionnel) : ce sont les conditions remplies après l'exécution du cas d'utilisation (état du système après réalisation du cas d'utilisation).
- **Exceptions** (optionnel) : un cas d'utilisation décrit le comportement du système lorsqu'il n'y a pas d'exception. Si une exception est levée, elle doit être décrite dans cette rubrique.

/// Client léger, riche et lourd

Né avec les technologies du Web, le client léger désigne un poste utilisateur dont la fonction se limite à interpréter l'affichage de pages web. Le client riche se limite à afficher les données mais en utilisant des API Java telles que Swing et nécessite un déploiement (Java Web Start). Issu des architectures client-serveur, le client lourd désigne un poste utilisateur (en Swing, par exemple) effectuant en plus de l'affichage, une part de traitements métier.

Lorsque le cas d'utilisation est lié à un acteur humain (« Gérer les clients », « Visualiser le catalogue »...), cela signifie que cet acteur a besoin d'interagir avec le système. Il faut donc lui associer une interface graphique. L'internaute et le client utilisent leur navigateur web pour accéder au système informatique (client léger), alors que les employés utilisent une application graphique déployée sur leurs postes (client riche).

Dans le cas où l'acteur serait un système externe (BarkBank ou PetEx), il n'y a pas d'interfaces graphiques. Les systèmes communiquent entre eux en échangeant des données dans un format pivot.

RETOUR D'EXPÉRIENCE Qui rédige les cas d'utilisation ?

Les cas d'utilisation relatent les besoins des utilisateurs. Il est donc normal que ce soit eux qui les rédigent. Malheureusement, ce n'est pas toujours le cas. En effet, même si les utilisateurs connaissent bien leur métier, ils ont bien souvent tendance à écrire très voire trop peu, persuadés que les analystes comprendront. Ainsi, la phrase anodine « Une fois les achats effectués, on obtient un bon de commande » peut susciter plusieurs interrogations, et notamment « Qu'est ce qu'un bon de commande ? », « Y a-t-il des contraintes légales pour certains produits ? », « Que fait-on du bon de commande ? »... Il est alors fréquent de rédiger les cas d'utilisation de manière bidirectionnelle, sur la base d'interviews et d'entretiens de recueil du besoin. Ainsi, un analyste posera des questions par écrit ou à l'oral à un utilisateur. Ce dernier y répondra, permettant ainsi à l'analyste de dresser les différents cas d'utilisation.

Gérer les clients

Résumé

Permet à un employé de créer/modifier/supprimer/rechercher/visualiser un client.

Acteurs

Employé.

Description

La société YAPS veut pouvoir créer ses clients dans le système à partir des données existantes. Elle souhaite également pouvoir les modifier, les supprimer et les rechercher. Les éléments caractérisant un client sont les suivants :

- identifiant unique du client ① ② ;
- login ① ② et mot de passe ② utilisés par le client pour se connecter à l'application ;
- prénom ② et nom de famille ② ;
- numéro de téléphone où l'on peut joindre le client et son adresse mail ;
- adresse postale : deux zones permettent de saisir l'adresse du client. La première est obligatoire ②, la deuxième optionnelle ;
- pays de résidence ②, ville ②, état et code postal ② ;
- date de naissance : YAPS veut pouvoir envoyer des cartes de vœux à la date d'anniversaire du client ;
- âge du client.

Une fois les données saisies, l'employé souhaite pouvoir les exploiter. Ainsi, à partir d'un identifiant, le système doit donner la possibilité

UML Les exceptions
dans les cas d'utilisation

Un cas d'utilisation décrit le comportement normal de l'application. Si des exceptions apparaissent, elles peuvent être référencées dans la description à l'aide de numéros ①, ②... Dans notre cas, il faut lire ces exceptions de la manière suivante : « le client à un identifiant unique, ① si cette valeur n'est pas unique, une exception est levée ; ② si cette valeur n'est pas renseignée alors qu'elle est obligatoire, une exception est levée ».

RETOUR D'EXPÉRIENCE Les maquettes

Les maquettes d'écrans facilitent la compréhension des cas d'utilisation. Souvent non informaticiens, les utilisateurs se repèrent facilement grâce à ce moyen visuel et peuvent entériner les choix émis par l'analyste.

d'afficher les coordonnées du client et proposer à l'employé de les mettre à jour ou de les supprimer. Dans le cas de la suppression, le système doit attendre une confirmation de l'employé avant de supprimer définitivement le client du système.

Le système doit aussi pouvoir afficher la totalité des clients présents dans le système.

Exceptions

- ① Valeur unique. Si cette donnée existe déjà dans le système, une exception doit être levée.
- ② Donnée obligatoire. Si cette donnée est manquante, une exception doit être levée.

Maquettes

Les employés de la société YAPS utilisent une application riche pour dialoguer avec le système. Pour la gestion des clients, ils utilisent un écran qui leur affiche la liste de tous les clients (menu *List customers*). Ils peuvent ensuite consulter les informations en cliquant sur le bouton *View* ou supprimer le client en cliquant sur *Delete*. Un autre menu (*Manage customer*) permet de manipuler les informations d'un client, c'est-à-dire la création, mise à jour, suppression et recherche à partir de son identifiant (figure 1–2).

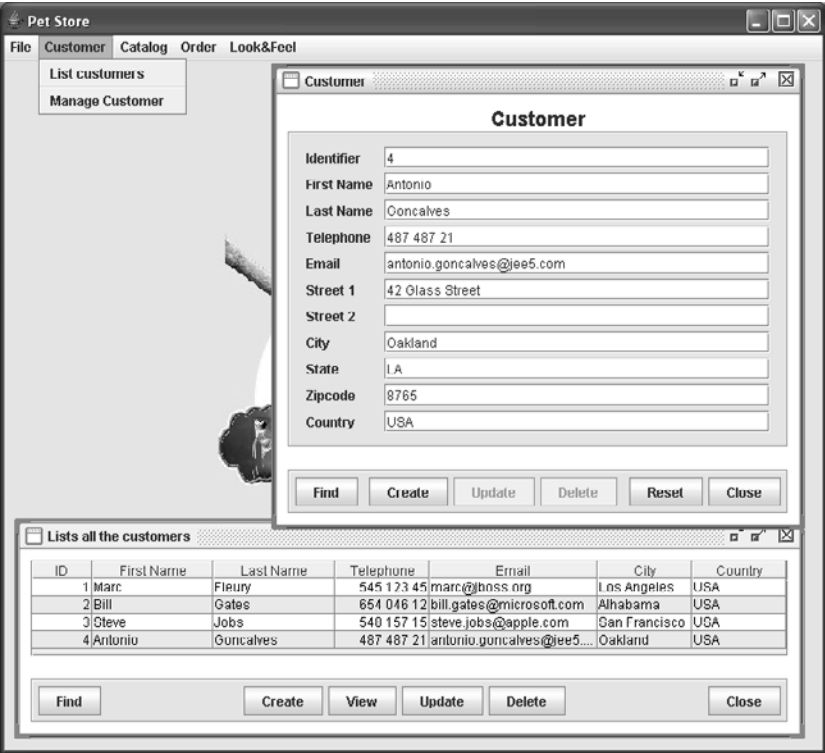


Figure 1–2
Application riche de gestion des clients

Gérer le catalogue

Résumé

Permet à un employé de créer/modifier/supprimer/rechercher/visualiser le catalogue des articles.

Acteurs

Employé.

Description

Le catalogue d'articles de la société YAPS est divisé en catégories. Bien qu'elle envisage d'étendre sa gamme, YAPS ne vend actuellement que cinq catégories d'animaux : poissons, chiens, chats, reptiles et oiseaux. Une catégorie est définie par les données suivantes :

- identifiant unique de la catégorie ❶ ❷ ;
- nom (exemple : Poisson, Chien, Chat...) ❷ ;
- description (exemple : un chien est un animal affectueux qui partagera avec vous des moments de bonheur) ❷.

Chacune de ces catégories est divisée en produits. Par exemple pour les chiens, on peut avoir les produits suivants : bulldog, caniche, dalmatien, labrador, lévrier. Chaque produit est défini comme suit :

- identifiant unique du produit ❶ ❷ ;
- nom (exemple : Bulldog, Caniche, Dalmatien...) ❷ ;
- description (exemple : un caniche est un petit chien affectueux qui ne prendra pas trop de place et saura vous réconforter par sa tendresse) ❷.

Enfin, chaque produit est, à son tour, divisé en articles. Ce sont ces articles qui sont proposés et vendus aux clients. Par exemple, le produit Caniche regroupe les articles suivants : caniche femelle adulte, caniche mâle adulte, caniche femelle 3 mois, caniche mâle 3 mois. Chaque article est défini comme suit :

- identifiant unique de l'article ❶ ❷ ;
- nom (exemple : Caniche 3 mois femelle...) ❷ ;
- prix unitaire de l'article ❷ ;
- image : donne une représentation graphique de l'article en question.

Exceptions

❶ Valeur unique. Si cette donnée existe déjà dans le système, une exception doit être levée.

❷ Donnée obligatoire. Si cette donnée est manquante, une exception doit être levée.

UML Granularité des cas d'utilisation

L'une des difficultés de la modélisation des cas d'utilisation consiste à trouver la bonne granularité. Un cas d'utilisation de trop grande taille signifie souvent qu'il peut être découpé en sous-cas, alors que de nombreux cas d'utilisation de petite taille peuvent être regroupés en un seul. Le cas « Gérer le catalogue » décrit de manière suffisante ce dont la société YAPS a besoin pour gérer son catalogue. Dans notre cas, il serait donc inutile et contre-productif d'avoir les cas d'utilisation « Créer une catégorie », « Mettre à jour une catégorie », « Supprimer une catégorie », « Créer un produit »...

Maquettes

L'application cliente riche de l'employé permet de gérer tous les éléments du catalogue, c'est-à-dire les catégories, les produits et les articles. Ci-après, les écrans permettant d'afficher la totalité du catalogue ainsi que de manipuler individuellement chacun des éléments le composant (figure 1–3).

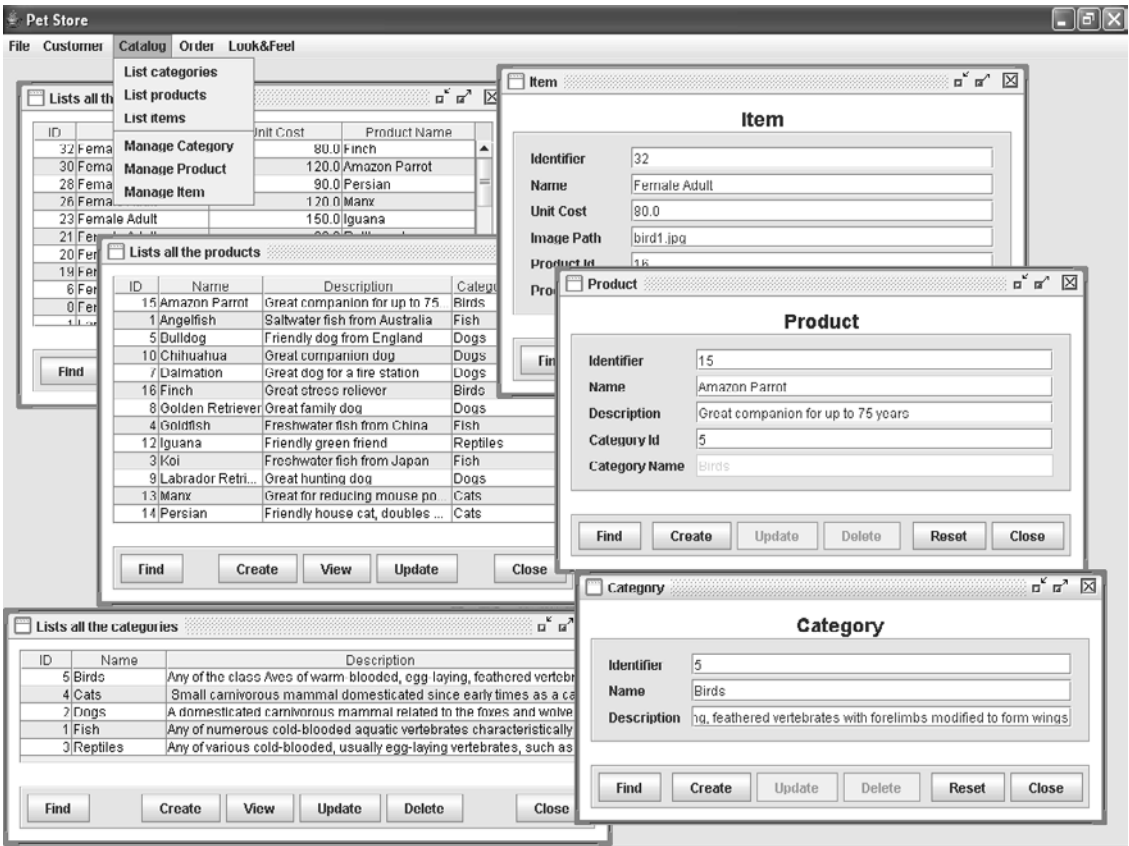


Figure 1–3 Application riche de gestion du catalogue

Visualiser les articles du catalogue

Résumé

Permet de visualiser le contenu du catalogue d'animaux domestiques.

Acteurs

Internaute, client.

Description

Les internautes et les clients peuvent visualiser la totalité du catalogue des animaux domestiques. L'organisation de l'affichage doit être intuitif.

tive, c'est-à-dire que le système doit afficher la liste des catégories, à partir desquelles le client choisit un produit puis un article.

Pour chaque article, une image représentant l'animal devra être affichée.

À tout moment, il doit être possible d'afficher les produits d'une catégorie différente.

Diagramme d'activités

Le diagramme d'activités ci-après nous donne la représentation graphique des actions effectuées pour visualiser le contenu du catalogue. Il doit être lu de la manière suivante : « Le système affiche les catégories du catalogue (chien, chat, poisson...). Lorsque l'internaute en sélectionne une, le système affiche les produits de la catégorie (bulldog, caniche...). Lorsqu'un produit est sélectionné, le système affiche les différents articles (bulldog femelle, caniche adulte...). Notez qu'à tout moment on peut revenir à l'action - Afficher les produits de la catégorie ».

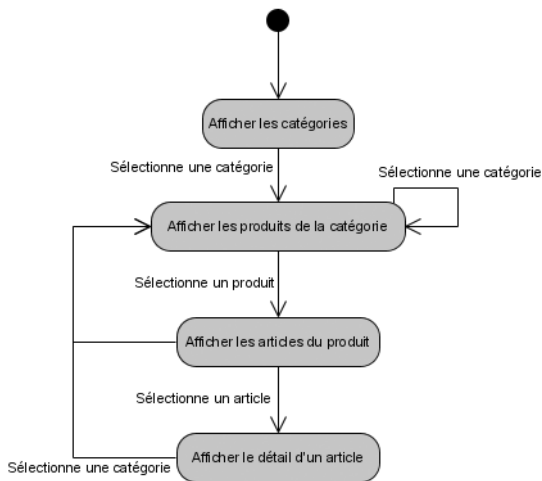


Figure 1-4
Diagramme d'activités de la visualisation
des articles du catalogue

UML Diagramme d'activités

UML permet de représenter graphiquement le comportement d'une méthode ou le déroulement d'un cas d'utilisation, à l'aide de diagrammes d'activités. Ce type de diagramme est utilisé pour représenter les aspects dynamiques d'un système à un niveau assez général. Il est composé d'un nœud initial (représenté par un point noir), d'activités liées entre elles par des événements, puis se termine par un nœud final (un rond noir entouré).

Maquettes

Les internautes et les clients visualisent le contenu du catalogue à partir de leur navigateur. Sur la colonne de gauche sont affichées les cinq catégories d'animaux domestiques vendus par la société YAPS. En cliquant sur la catégorie *Dogs* (chiens), l'internaute est redirigé vers une page qui affiche les produits de cette catégorie. Pour chaque produit, on affiche son nom et sa description (figure 1-5).

En cliquant sur le produit *Bulldog*, l'internaute est redirigé vers la liste des articles. Dans l'exemple ci-après, ce produit possède deux articles : un mâle et une femelle. Pour chaque article, on affiche son nom et son prix (figure 1-6).

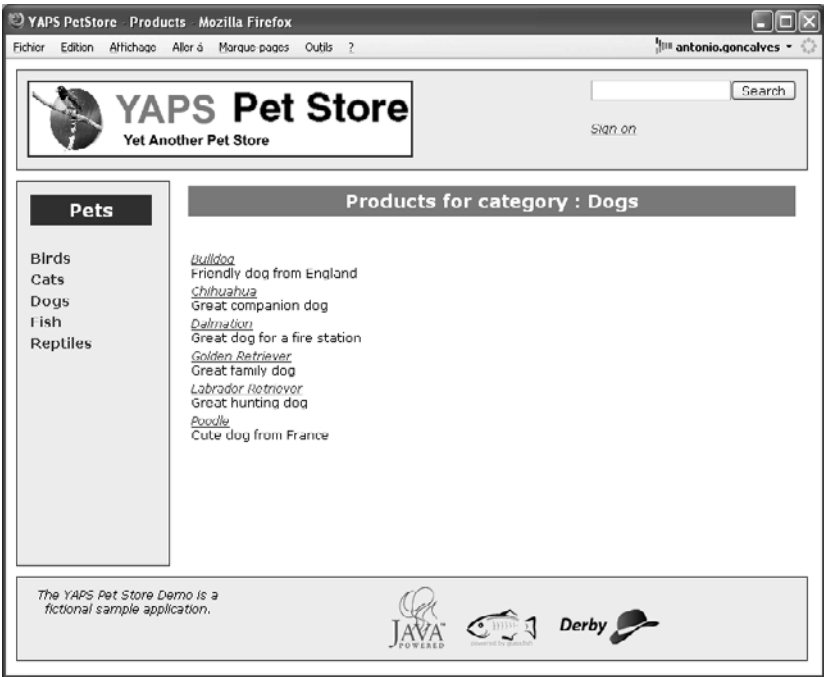


Figure 1-5
Affichage de tous les produits
de la catégorie Dogs



Figure 1-6
Affichage de tous les articles
du produit Bulldog

Enfin, pour connaître le détail d'un article, il suffit de cliquer sur son nom pour arriver sur la page de description. Le nom et le prix de l'article sont affichés ainsi que l'image représentant l'animal (figure 1-7).



Figure 1–7
Affichage du détail d'un article

Rechercher un article

Résumé

Permet de rechercher un article par son nom ou le nom de son produit.

Acteurs

Internaute, client.

Description

En plus de visualiser le catalogue de manière linéaire (voir cas d'utilisation « Visualiser les articles du catalogue »), les internautes et les clients peuvent rechercher les animaux domestiques contenus dans le système à partir d'une chaîne de caractères.

Par exemple, si la chaîne de caractères saisie est « iche » le système retournera les articles suivants :

Nom de l'article	Nom du produit
Caniche nain mâle adulte	Caniche
Femelle 3 mois	Caniche
Plus petit qu'un caniche	Chihuahua

La recherche ne tient pas compte des minuscules ou majuscules. Si aucun article ne correspond aux critères demandés, une information est affichée à l'internaute pour lui indiquer que sa recherche n'a pas abouti et qu'il doit modifier le critère de recherche.

Maquettes

Pour rechercher les articles, l'internaute utilise la zone de saisie située dans l'en-tête de toutes les pages du site. Cette zone est suivie d'un bouton *Search*. Lorsque l'internaute clique sur ce bouton après avoir saisi un texte, le système retourne la liste des articles qui répondent au critère. Par exemple, ci-après, la liste des articles répondant au critère *adult* (figure 1–8).

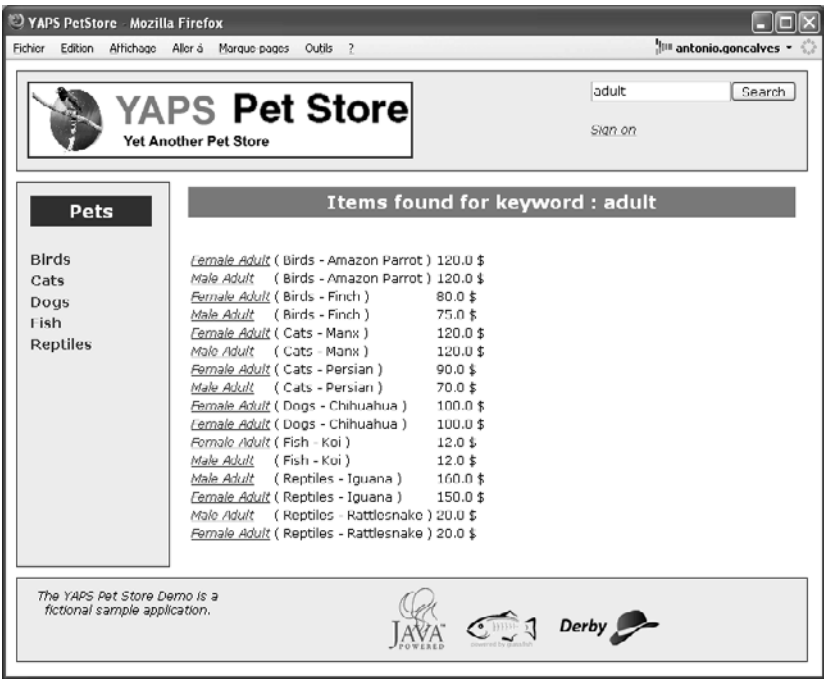


Figure 1–8
Liste des articles répondant
au critère de recherche

Se créer un compte

Résumé

Permet à un internaute de se créer un compte dans le système et de devenir ainsi un client.

Acteurs

Internaute.

Description

Ce cas d'utilisation diffère du cas « Gérer les clients » dans le sens où l'internaute ne peut renseigner que ses propres données. Pour se créer un compte, l'internaute doit saisir un login ❶, un mot de passe et ressaisir une seconde fois son mot de passe ❷. Le système lui demande alors de saisir ses coordonnées et informations personnelles (identiques à celles du cas d'utilisation « Gérer les clients »).

Exceptions

- ❶ Le login doit être unique dans le système. Si ce n'est pas le cas, l'internaute doit en être averti et doit en choisir un autre.
- ❷ Si les deux mots de passe ne sont pas identiques, une exception doit être levée.

Post-conditions

L'internaute est connu du système, il devient client de la société YAPS.

Maquettes

Pour se créer un compte, l'internaute clique sur le menu *Sign on*, puis saisit un login unique suivi de deux fois son mot de passe. Après vérification de la validité des mots de passe et de leur concordance, le système lui demande de compléter ses informations (figures 1-9 et 1-10).

UML Post-conditions des cas d'utilisation

Les post-conditions représentent l'état (les résultats) du cas d'utilisation à la fin de son exécution. Si le cas d'utilisation « Se créer un compte » se déroule normalement, l'internaute devient un client de la société YAPS.

Figure 1-9

Le client saisit son login et deux fois son mot de passe.

Figure 1-10
Saisie des informations du client

YAPS Pet Store
Yet Another Pet Store

Search

Account Cart Sign off

Create Your Account

Personal information

Login : james

*Firstname : James

*Lastname : Rorisson

Email : jr@mymail.com

Telephone : 02154 254 147

Date of birth (dd.MM.yyyy): 24.12.1968

Address

*Street1 : Ritherdon Road

Street2 :

*City : New York

State :

*Zipcode : 4568

*Country : USA

Submit

Se connecter et se déconnecter

Résumé

Permet à un client de se connecter et de se déconnecter du système.

Acteurs

Client.

Pré-conditions

Le client s'est auparavant créé un compte (cas d'utilisation « Se créer un compte »).

Description

Le client saisit son login et son mot de passe ❶ ❷. Il est reconnu par le système, qui affiche alors son nom et prénom. Lorsque le client se déconnecte, il redevient internaute jusqu'à sa prochaine connexion.

Exceptions

- ❶ Si le login n'est pas connu du système, une exception doit être levée.
- ❷ Si le mot de passe n'est pas le bon, une exception doit être levée.

UML Pré-conditions des cas d'utilisation

Pour exécuter un cas d'utilisation, les pré-conditions doivent être remplies. Dans l'exemple du cas d'utilisation « Se connecter et se déconnecter », le client doit auparavant s'être créé un compte pour pouvoir se connecter à l'application.

Maquettes

En cliquant sur le lien *Sign on*, l'internaute est redirigé vers une page lui demandant de s'authentifier (figure 1-11). Après avoir saisi son identifiant et son mot de passe, il est dirigé vers la page d'accueil.

Figure 1-11
Saisie du login et du mot de passe

Figure 1-12
La page d'accueil affiche le nom et prénom du client.

Cette fois, la page d'accueil affiche le nom et prénom du client, ainsi que trois liens lui permettant de se déconnecter (*Sign Off*), de consulter ses informations (*Account*) et de visualiser le contenu de son panier électronique (*Cart*) (figure 1-12).

Consulter et modifier son compte

Résumé

Permet à un client de consulter et de mettre à jour ses informations personnelles dans le système.

Acteurs

Client.

Pré-conditions

Le client doit être connecté au système (cas d'utilisation « Se connecter et se déconnecter »).

Description

Ce cas d'utilisation diffère du cas « Gérer les clients » dans le sens où le client ne peut consulter et modifier que ses données personnelles. Celles-ci sont identiques à celles du cas d'utilisation « Gérer les clients ».

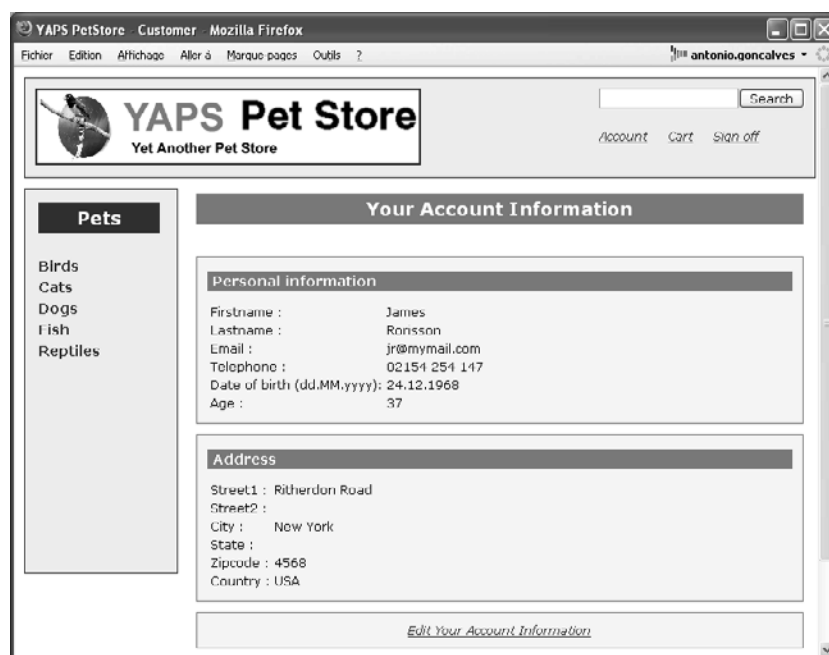


Figure 1-13
Les coordonnées du client
s'affichent en lecture seule.

Maquettes

Une fois connecté, le client peut consulter ses informations en cliquant sur le lien *Account*. Cette page de consultation affiche, en lecture seule, les informations du client. Il peut ensuite les modifier en cliquant sur le lien *Edit your account information* (figures 1–13 et 1–14).

Figure 1–14

Le client peut mettre à jour ses coordonnées.

Acheter des articles

Résumé

Permet à un client d'acheter des articles.

Acteurs

Client.

Pré-conditions

Le client doit être connecté au système (cas d'utilisation « Se connecter et se déconnecter »).

Description

Un client visualise le catalogue (voir cas d'utilisation « Visualiser les articles du catalogue ») ou recherche un animal domestique (voir cas d'utilisation « Rechercher un article »). Lorsqu'il est intéressé par un article, il

lui suffit de cliquer sur un lien pour ajouter cet article dans son panier électronique (Caddie). Cette opération peut être exécutée plusieurs fois sur des articles différents. Le client a ensuite la possibilité de modifier la quantité désirée pour chaque article ou supprimer un ou plusieurs de ces articles du panier. Lorsque la quantité d'un article est égale (ou inférieure) à zéro, l'article est automatiquement supprimé du panier.

Pendant toute la durée de sa session, le client peut visualiser le contenu de son panier quand bon lui semble. Lorsque le Caddie est vide, un message avertit le client. Sinon, le système affiche la liste des articles avec le nom, la description du produit, la quantité désirée, le prix unitaire et le sous-total (prix unitaire $\times$ quantité). Le montant total du panier est également renseigné. Ce Caddie est illimité en taille, un client peut donc acheter autant d'articles qu'il le souhaite.

Lorsque le client est satisfait, il valide son panier électronique. Il doit alors saisir les informations de sa carte bancaire ainsi que l'adresse de livraison. Par défaut, l'adresse de livraison est la même que celle du client mais elle peut être modifiée. Les données de la carte bancaire sont les suivantes :

- Numéro de carte bancaire.
- Type de carte bancaire (Visa, Master Card et American Express).
- Date d'expiration de la carte bancaire. Le format de cette date est MM/AA, c'est-à-dire deux chiffres pour le mois et deux pour l'année, séparés par le caractère /.

Une fois toutes ces données validées ❶, un bon de commande est créé. Le panier électronique est alors automatiquement vidé.

Exceptions

❶ Les données de la carte bancaire sont validées par BarkBank. Si la banque rejette la carte bancaire, le client en est averti et peut ressaisir ses données.

Post-condition

Exécuter le cas d'utilisation « Créer un bon de commande ».

Maquettes

Lorsque l'internaute s'authentifie, le menu *Cart* apparaît en haut de la page. Ce lien permet d'afficher le contenu du panier électronique. Si ce dernier est vide, la page affiche un message avertissant le client : *The shopping Cart is empty* (figure 1–15).

Pour remplir le panier, il suffit de se rendre sur la page de description des articles et de cliquer sur le lien *Add to cart*. Cette action ajoute dans le Caddie l'article sélectionné avec une quantité égale à un (figure 1–16).



Figure 1–15
Le panier électronique est vide.



Figure 1–16
Le client ajoute des articles
en cliquant sur Add to cart.

Après avoir effectué différents achats, le client clique sur le lien *Cart* pour consulter le contenu de son panier électronique. Cette page affiche le nom des articles achetés ainsi que leur quantité et leur prix. Le client peut à tout moment modifier la quantité de chaque article en cliquant sur *Update* ou supprimer un article en cliquant sur *Remove*. En bas du tableau s’affiche le montant total du panier électronique (figure 1–17).

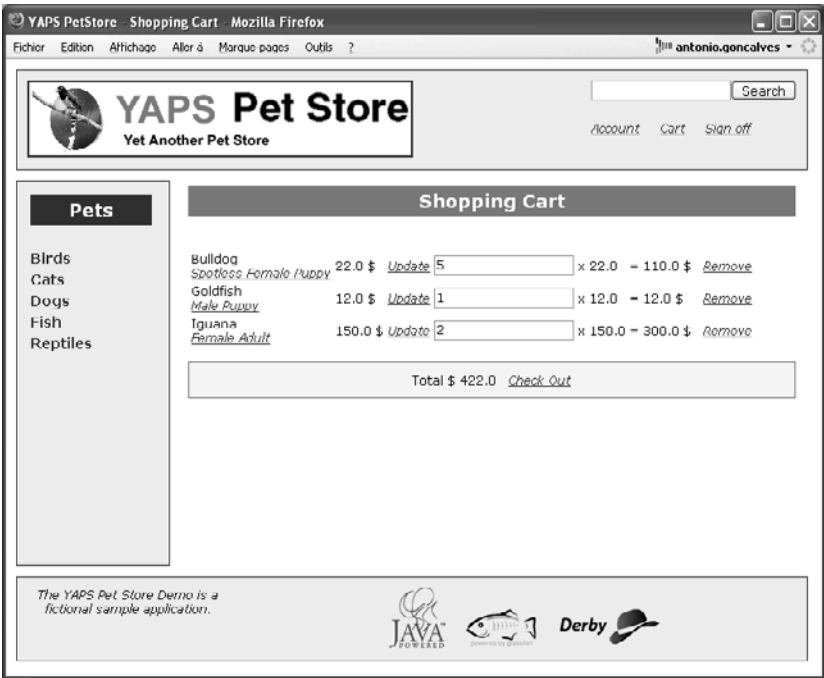


Figure 1–17
Contenu du panier électronique

Une fois les achats terminés, le client clique sur le lien *Check out*. Cette action l’amène sur une page lui demandant de saisir l’adresse de livraison et les coordonnées de sa carte bancaire (figure 1–18).

Le client valide la page en cliquant sur *Submit*. Il est alors redirigé vers une page qui lui confirme la création de sa commande et lui en donne le numéro ainsi que son récapitulatif (figure 1–19).

YAPS Pet Store
Yet Another Pet Store

Search

Account Cart Sign off

Pets

Birds
Cats
Dogs
Fish
Reptiles

Confirm Order

Personal information

Firstname : James
Lastname : Harrison
Email : jhr@mymail.com

Delivery Address

*Street1 : Ritherdon Road
Street2 :
*City : New York
State :
*Zipcode : 4568
*Country : USA

Credit Card

*Credit card number : 1234 5678 9101
*Type : Visa
*Expiry date (YY/MM): 00/00

Submit

Figure 1–18
Saisie de l'adresse de livraison
et du mode de paiement

YAPS PetStore Items Mozilla Firefox

Search

Account Cart Sign off

Pets

Birds
Cats
Dogs
Fish
Reptiles

Your Order is Complete : 103

Your order id is 103

Bulldog Spotless Female Puppy	22.0 \$	5	x 22.0 = 110.0 \$
Goldfish Male Puppy	12.0 \$	1	x 12.0 = 12.0 \$
Iguana Female Adult	150.0 \$	2	x 150.0 = 300.0 \$

Total \$ 422.0

You will receive shortly an email confirming your order
Thank you for shopping with the YAPS Pet Store

The YAPS Pet Store Demo is a fictional sample application.

JAVA POWERED Derby

Figure 1–19
Confirmation de la création
du bon de commande

Créer un bon de commande

Résumé

Une fois le panier électronique validé par le client, un bon de commande est créé.

Acteurs

Client.

Pré-conditions

Le client achète des articles et valide son panier électronique (voir cas d'utilisation « Acheter des articles »).

Description

Lorsque le panier électronique du client est validé, le système crée automatiquement un bon de commande. Ce dernier contient toutes les informations nécessaires pour être traité :

- un numéro de bon de commande ;
- la date de création de ce bon de commande ;
- les références du client qui a acheté les articles ;
- les lignes de commande : une ligne de commande référence l'article acheté et sa quantité. Il y a autant de lignes de commande que d'articles contenus dans le panier électronique ;
- les informations de la carte bancaire ;
- l'adresse de livraison des animaux.

Cette création du bon de commande entraîne plusieurs traitements :

- 1** Le bon de commande est imprimé puis stocké dans les archives de la société YAPS.
- 2** Toutes les informations nécessaires à l'acheminement des animaux sont envoyées au transporteur PetEx de manière électronique au format XML. PetEx livre ensuite les animaux aux nouveaux heureux propriétaires.
- 3** Un e-mail est envoyé au client pour l'informer du bon déroulement de sa transaction. Cet e-mail contient le numéro du bon de commande ainsi qu'un récapitulatif de son contenu.
- 4** Pour des raisons légales, les employés doivent être avertis des bons de commande contenant des reptiles (une alerte s'affiche dans l'interface graphique de l'employé).

Visualiser et supprimer les commandes

Résumé

Permet à un employé de visualiser et de supprimer les commandes présentes dans le système.

Acteurs

Employé.

Description

L'employé peut visualiser la liste des commandes présentes dans le système. Pour chacune de ces commandes, il peut en consulter les informations et les supprimer.

Pour des raisons légales, les employés de YAPS veulent être avertis en temps réel des achats de reptiles. Selon les pays, il faut en effet avertir les autorités. Ainsi, dès qu'une commande contenant des reptiles est passée, les employés en sont avertis par une alerte qui s'affiche dans l'interface graphique.

Maquettes

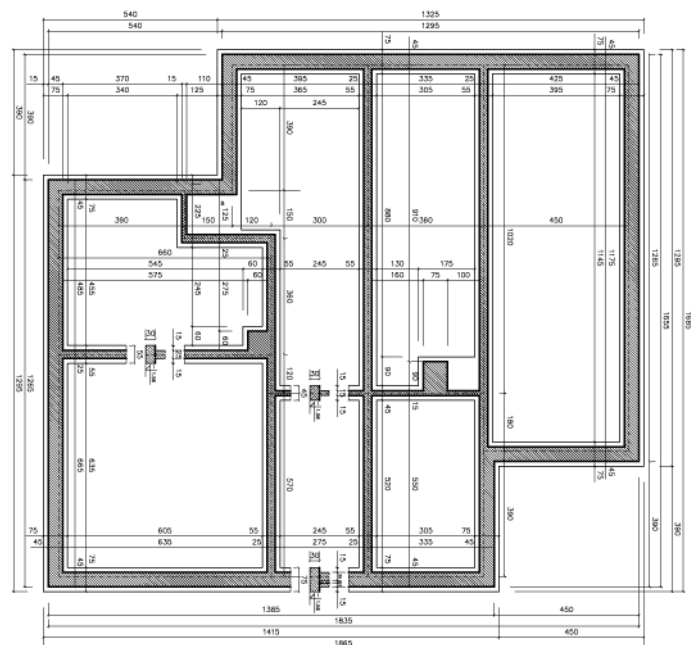


Figure 1–20
Application client riche
de la gestion des bons de commande

En résumé

Dans ce chapitre, nous avons présenté l'étude de cas de l'application YAPS Pet Store ainsi que les acteurs qui l'utilisent. Le diagramme de cas d'utilisation nous a permis de formaliser les besoins de manière synthétique, puis d'explicitier chaque cas d'utilisation de manière textuelle. Les diagrammes d'activités et les maquettes d'écrans sont venus éclaircir la dynamique de certains cas d'utilisation. Cette application sera conçue et réalisée au cours des prochains chapitres.

chapitre



Architecture de l'application

Dans le précédent chapitre, nous avons décrit le comportement souhaité de la future application de commerce électronique de la société YAPS. Vous allez maintenant découvrir les technologies utilisées pour développer cette application, c'est-à-dire le langage Java et la nouvelle plate-forme Java EE 5. Grâce à l'utilisation de diagrammes UML de composants et de déploiement, nous allons modéliser et décrire l'architecture logicielle de l'application YAPS Pet Store. Celle-ci s'inspire du Blueprint Java Pet Store de Sun et de ses design patterns. Elle est architecturée en couches (présentation, traitements et accès aux données) et utilise la plate-forme Java EE 5 qui s'appuie sur les nouveautés du langage Java 5 (annotations, génériques...).

SOMMAIRE

- ▶ Technologies utilisées
- ▶ Nouveautés du langage Java 5
- ▶ La plate-forme Java EE 5
- ▶ Les Blueprints et le Pet Store de Sun
- ▶ Architecture en couches

MOTS-CLÉS

- ▶ Java SE
- ▶ Java EE
- ▶ EJB
- ▶ JPA
- ▶ JMS et MDB
- ▶ JSP, JSTL
- ▶ JSF
- ▶ XML
- ▶ Web Service
- ▶ Design pattern
- ▶ UML

Présentation des langages utilisés

Java SE 5

Avant de parler de Java Enterprise Edition 5 (Java EE 5), il est nécessaire de présenter brièvement le langage sur lequel s'appuie cette plate-forme : Java 5.0. La version 5 de Java SE, ou *Java Standard Edition*, est une révision majeure du langage Java créé en 1995 par l'équipe de James Gosling. Cette version apporte de nombreuses nouveautés telles que l'autoboxing, les annotations, les génériques, une nouvelle boucle d'itération, les types énumérés, les méthodes à arguments variables, les imports statiques et bien d'autres encore. De nouveaux outils ainsi que de nouvelles API ont vu le jour ou ont été considérablement enrichis, comme l'API de concurrence, l'API de thread, la supervision de la JVM, etc.

API

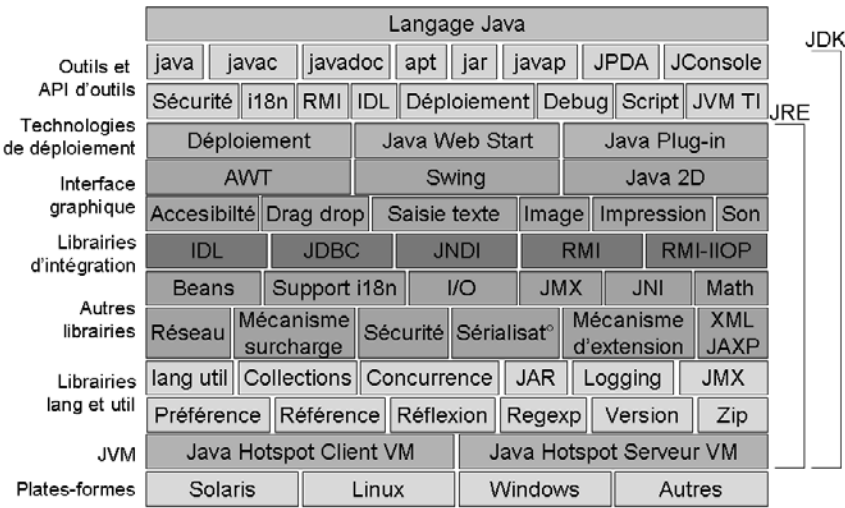
Une API, ou *Application Programming Interface*, définit la manière dont un composant informatique peut être invoqué par un autre. Il s'agit généralement d'une liste de méthodes que l'on peut appeler.

À LIRE Java 5

Il existe beaucoup de livres sur le langage Java ainsi que de nombreuses références et articles en ligne, notamment :

- Emmanuel Puybaret, *Java 1.4 et 5.0*, Eyrolles, 2006
 - Claude Delannoy, *Programmer en Java*, Eyrolles, 2007
 - Anne Tasso, *Le livre de Java premier langage*, Eyrolles, 2006
- <http://java.sun.com/docs/books/jls/>

Figure 2-1
L'architecture de Java SE 5



La figure 2-1 montre les composants qui constituent Java SE 5. Cet ouvrage n'a pas la prétention de vous expliquer toutes les nouveautés du langage. Toutefois, il est important de s'attarder sur celles qui seront utilisées tout au long de notre étude de cas. La suite de cette partie tend à présenter succinctement les éléments caractéristiques de Java SE 5 auxquels nous allons nous confronter.

Autoboxing

Le langage Java s'appuie sur des types primitifs pour décrire les types de base (byte, short, int, long, double, float, boolean et char). Toutefois, comme tout n'est qu'objet en Java, il est nécessaire de constamment

encapsuler ces types primitifs dans des classes de référence (Integer pour int, Double pour double, etc.).

Ce type de transformation (appelé *boxing*) peut rapidement s'avérer pénible. D'autant que le processus inverse (appelé *unboxing*) est nécessaire pour retrouver son type primitif initial. Avec Java 5, cette conversion explicite devient obsolète puisque l'autoboxing convertit de manière transparente les types de base en références et réciproquement. Bien sûr, vous pouvez continuer à utiliser uniquement les types primitifs si vous le souhaitez.

Exemple d'autoboxing

```
// Boxing explicite entre un Integer et un int
Integer objet = new Integer(5);
int primitif = objet.intValue();

// Autoboxing
Integer objet = 5;
int primitif = objet;
```

Annotations

Une annotation permet de marquer (on dit alors *annoter*) certains éléments du langage Java afin de leur ajouter une propriété particulière. Ces annotations peuvent ensuite être utilisées à la compilation ou à l'exécution pour automatiser certaines tâches. Une annotation peut être utilisée sur plusieurs types d'éléments (paquetage, classe, interface, énumération, annotation, constructeur, méthode, paramètre, attribut de classe ou variable locale).

Exemple d'utilisation d'annotations

```
@CeciEstUneAnnotationSurUneClasse
public class MaClasse {

    @UneAnnotationSurUnAttribut
    private Date unAttribut;

    @SurUneMethode
    private void maMethode() {
        return;
    }
}
```

Comme vous le verrez tout au long de cet ouvrage, Java Enterprise Edition 5 utilise très fréquemment les annotations. Nous aurons l'occasion de nous y attarder plus longuement par la suite.

JAVA 5 JConsole

La JConsole est un utilitaire de surveillance fourni avec Java SE 5. Liée aux technologies JMX et MBean, la JConsole permet de surveiller et superviser les applications Java (occupation mémoire, threads en cours, classes chargées...) tout comme de prendre en charge certaines opérations (appeler le garbage collector, changer le niveau des logs, etc.).

INFORMATION Acronymes

La plate-forme Java est extrêmement riche. Elle a donc tendance à utiliser abondamment et à abuser d'acronymes en tout genre (souvent commençant par la lettre « J »). Vous trouverez en annexe un lexique d'acronymes et de sigles.

APPROFONDIR Annotations et génériques

Si vous voulez en savoir plus sur les annotations et les génériques, consultez le tutoriel de Sun. Vous y trouverez une information mise à jour ainsi que des exemples de code.

► <http://java.sun.com/docs/books/tutorial>

APPROFONDIR Les types énumérés

Les types énumérés offrent d'autres possibilités : itération, utilisation dans un `switch`, `EnumSet`, `EnumMap`, etc. Pour en savoir d'avantage, consultez le site de Sun à l'adresse :

► <http://java.sun.com/j2se/1.5.0/docs/guide/language/enums.html>

RAPPEL Périmètre de ce livre

Ce livre n'a pas la prétention de vous enseigner les nouveautés du langage Java mais uniquement de vous présenter celles qui vont être utilisées lors de la réalisation de l'application YAPS Pet Store. Si vous n'êtes pas encore à l'aise avec les annotations, les génériques ou les types énumérés, reportez-vous aux références données dans cet ouvrage pour approfondir vos connaissances.

Génériques

Pour les personnes familières des templates C++, les génériques sont simples à comprendre, même si leur fonctionnement n'est pas du tout similaire. Ils permettent de ne pas spécifier le type à la compilation (paramètre ou retour de méthode, par exemple), tout en assurant que le type reste cohérent dans ses différentes utilisations. Il est par exemple possible de spécifier qu'une collection (un objet `Vector`, `Hashtable` ou `Array`, par exemple) ne peut être remplie que par un type de classe donné (`Vector<Integer>` pour déclarer un vecteur d'entiers). Il n'est donc plus nécessaire d'effectuer le contrôle du type au moment de l'exécution.

Exemple d'un vecteur générique

```
// Sans générique
Vector nombres = new Vector();
for (int i = 0; i < nombres.size(); i++) {
    Integer nombre = (Integer) nombres.elementAt(i);
}

// Avec générique
Vector <Integer> nombres = new Vector<Integer>();
for (int i = 0; i < nombres.size(); i++) {
    Integer nombre = nombres.elementAt(i);
}
```

Comme on peut le constater dans le fragment de code ci-dessus, le parcours des éléments du vecteur avec générique présente une meilleure lisibilité ainsi qu'une plus grande robustesse.

Les types énumérés

Java 5.0 introduit le nouveau mot-clé `enum`, utilisable comme le mot-clé `class`. Sa particularité est qu'il représente un type énuméré, c'est-à-dire un type qui n'accepte qu'un ensemble fini d'éléments. Il peut donc être utilisé pour définir un ensemble de constantes.

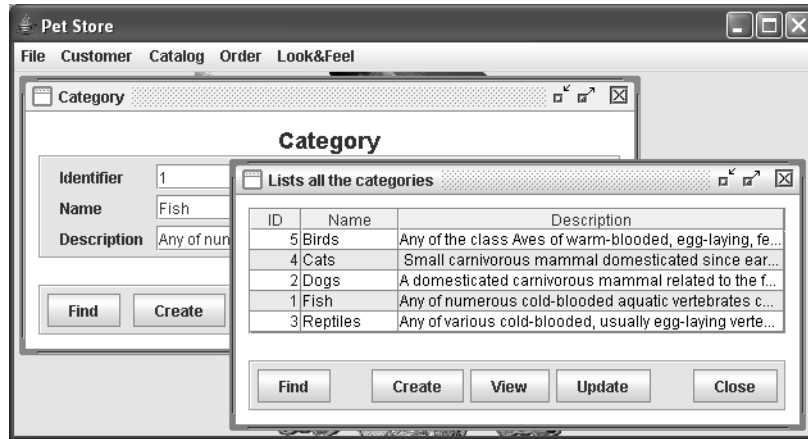
Exemple d'une énumération

```
public enum Saisons {
    PRINTEMPS,
    ETE,
    AUTOMNE,
    HIVER
};
```

Swing

La plupart des applications destinées aux utilisateurs comportent des éléments graphiques de telle sorte qu'elles soient plus conviviales et

ergonomiques. La plate-forme Java dispose des API AWT et Swing permettant de construire des interfaces homme-machine (IHM) sophistiquées en client lourd.



À LIRE **Swing**

Pour de plus amples informations, reportez-vous aux références suivantes :

- 📖 Emmanuel Puybaret, *Swing*, Eyrolles, 2006
- 📖 Kathy Walrath, *The JFC Swing Tutorial*, Addison-Wesley, 2004

Figure 2–2
Application cliente développée en Swing

Swing sera utilisé dans cet ouvrage pour développer une partie de l'interface utilisateur. Toutefois, cette bibliothèque très riche, et parfois complexe, ne sera pas enseignée dans ce livre.

JNDI 1.5

Ce composant, appelé communément *service de nommage*, est un service fondamental dans n'importe quel système informatique. Il permet d'associer des noms à des objets et de retrouver ces objets grâce à leurs noms. *Java Naming and Directory Interface* (JNDI) fournit les fonctionnalités de nommage et d'annuaire aux applications écrites en Java.

Omniprésent dans la version J2EE 1.4, JNDI se fait plus discret et il est intégré directement dans Java SE 5. Il continue à être une pièce maîtresse mais de manière plus transparente.

JDBC 3.0

JDBC (*Java Data Base Connectivity*) est une API permettant l'accès à des bases de données relationnelles à partir du langage Java. Elle se charge de trois étapes indispensables à l'accès aux données :

- la création d'une connexion à la base ;
- l'envoi d'instructions SQL ;
- l'exploitation des résultats provenant de la base.

Pour accéder à la base de données, JDBC s'appuie sur des drivers (pilotes) spécifiques à un fournisseur de SGBDR ou sur des pilotes génériques.

Dans le chapitre 6, nous utiliserons JNDI pour accéder aux composants distants à partir de l'interface Swing.

APPROFONDIR **JNDI**

Java Naming and Directory Interface est rarement utilisé seul. Il est généralement employé avec les EJB. Il n'y a donc que peu de livres s'attardant uniquement sur cette API.

- 📖 Rosanna Lee, *The Jndi Api: Tutorial and Reference: Building Directory-Enabled Java Applications*, Addison-Wesley, 2000
- ▶ <http://java.sun.com/products/jndi/>

PERSISTANCE **Les pilotes JDBC**

Les pilotes JDBC sont classés en quatre catégories :

- Les pilotes de type 1 (pont JDBC/ODBC) permettent de convertir les appels JDBC en appel ODBC (*Open Database Connectivity*),
- Les pilotes de type 2 sont écrits en code natif et dépendent de la plate-forme.
- Les pilotes de type 3 utilisent un autre pilote JDBC intermédiaire pour accéder à la base de données.
- Les pilotes de type 4 sont écrits entièrement en Java. Ils sont donc portables.

▄ Les balises

Une balise est une portion de texte encadré par les caractères < et >. Elle sert à délimiter des ensembles de données contenues dans un document afin de le structurer. XML est un langage de balisage.

APPROFONDIR XML et XSD

- 📖 Alexandre Brillant, *XML - Cours et exercices*, Eyrolles, 2007
- 📖 Antoine Lonjon, Jean-Jacques Thomasson, Libero Maesano, *Modélisation XML*, Eyrolles, 2006
- 📖 Mitch Amiano, Conrad D'Cruz, Michael D. Thomas, Kay Ethier, *XML*, Wrox, 2006
- ▶ <http://www.w3.org/XML/>
- ▶ <http://www.w3.org/XML/Schema>

APPROFONDIR HTML/XHTML

- 📖 Mathieu Nebra, *Réussir son site web avec XHTML et CSS*, Eyrolles, 2008
- 📖 Jean Engels, *XHTML et CSS*, Eyrolles, 2006
- 📖 Raphaël Goetter, *Mémento XHTML*, Eyrolles, 2006
- ▶ <http://www.w3.org/MarkUp/>
- ▶ <http://www.w3.org/TR/xhtml1/>

XML et XSD

SGML (*Standard Generalized Markup Language*, ou langage normalisé de balisage généralisé), adopté comme standard en 1986, a été la première tentative pour créer des documents électroniques. L'idée principale était de séparer le contenu d'un document de sa forme. SGML a été une innovation, mais d'une complexité telle que sa manipulation s'est trouvée restreinte aux spécialistes.

XML (*eXtensible Markup Language*, ou langage extensible de balisage), issu de SGML, a été mis au point par le World Wide Web Consortium (W3C) en 1996. Contrairement à HTML, qui présente un jeu limité de balises orientées présentation (titre, paragraphe, image, lien hypertexte...), XML est un métalangage, qui va permettre d'inventer à volonté de nouvelles balises pour décrire des données et non leur représentation.

XML permet donc de définir des fichiers dont la structure est personnalisée par la création de balises. De fait, ce langage s'impose comme un standard dans les échanges inter-systèmes d'information. XML devient un format pivot, qualifié encore de format d'échanges. De plus, un certain nombre d'API offre des mécanismes pour créer, extraire et vérifier la validité d'un document XML. Cette validation n'est possible que si l'on connaît la structure du document. Cette structure est définie par un *XML Schema Definition* (XSD), technologie dérivée d'XML. Un schéma XML (XSD) est lui-même un fichier XML.

Exemple de document XML

```
<racine>
  <titre nom='exemple de message XML' />
  <message>
    données envoyées entre émetteur et récepteur
  </message>
</racine>
```

HTML et XHTML

À partir de 1992, Internet popularise le langage HTML (*Hypertext Markup Language*, ou langage de balisage hypertexte, conçu vers 1990) pour la présentation de documents électroniques hypertextes. Issu de SGML, HTML définit un certain nombre de balises liées uniquement à la présentation. Depuis quelques années, le HTML tend à être remplacé par le XHTML qui lui apporte la rigueur de la notation XML.

Exemple de page HTML

```

<html>
<head>
  <title>Page HTML affichant Hello World</title>
</head>
<body>
  <center>Hello World</center>
</body>
</html>

```

La plate-forme Java EE 5

Java EE, ou Java Enterprise Edition, est un ensemble de spécifications destinées aux applications d'entreprise. Java EE peut être vu comme une extension du langage Java afin de faciliter la création d'applications réparties, robustes, performantes et à haute disponibilité.

Comme beaucoup, je pense que Java EE est aujourd'hui la meilleure plate-forme de développement pour les entreprises. Elle combine les avantages du langage Java avec l'expérience acquise dans le développement au cours des dix dernières années. Elle bénéficie en outre du dynamisme des communautés Open Source ainsi que du JCP de Sun.

Rappel Java EE 5 dans cet ouvrage

La nouvelle plate-forme Java EE 5 comporte plus de vingt spécifications (voir annexe A). Il est impossible en un seul ouvrage de couvrir toutes les particularités de ces spécifications. Le développement de notre étude de cas nous permettra d'utiliser les plus importantes et surtout de voir comment elles s'utilisent ou interagissent les unes par rapport aux autres. Pour approfondir vos connaissances, n'hésitez pas à consulter les nombreuses références contenues dans ce livre.

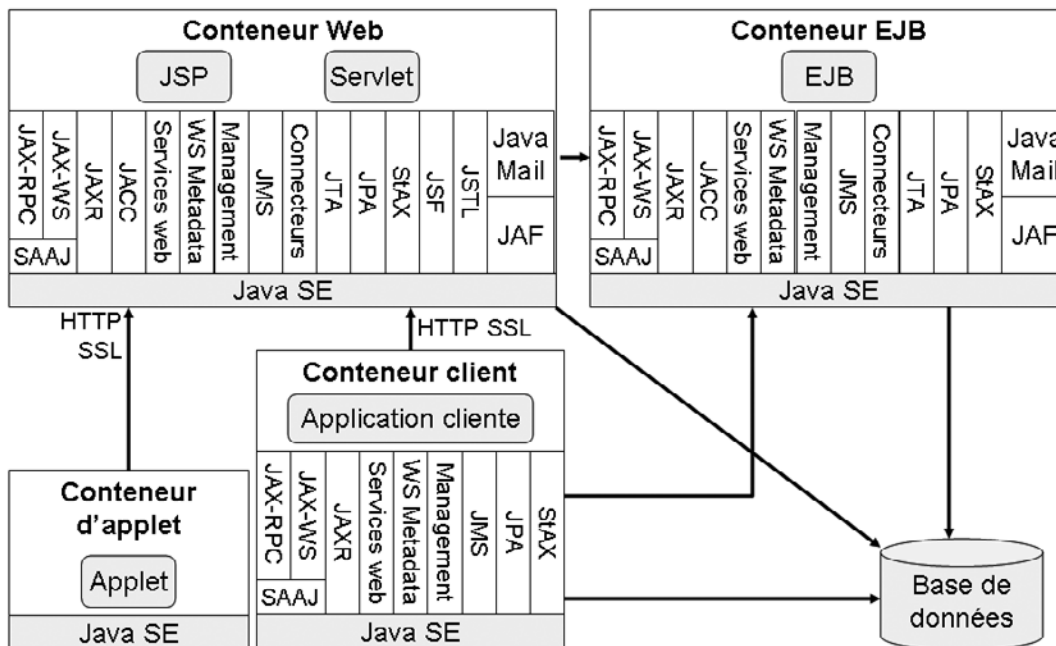


Figure 2-3 L'architecture de Java EE 5

/// JCP

Java Community Process est le processus utilisé par Sun et de nombreux partenaires pour coordonner l'évolution du langage Java et des technologies associées.

PRÉCISION Conteneur client

Le conteneur client, ou *Application Client Container* (ACC), apporte aux applications Java SE (par exemple, Swing) des services de sécurité, de nommage, d'injection...

PRÉCISION J2EE, Java EE, J2SE, Java SE

Pendant longtemps, Java et sa plate-forme entreprise ont été appelés respectivement J2SE et J2EE. Depuis la version 5, le chiffre « 2 » a disparu pour faciliter la compréhension de la version utilisée. Ce livre utilisera donc les nouvelles appellations Java SE et Java EE. Le terme J2EE sera utilisé pour désigner l'ancienne spécification.

JPA, Java Persistence API, est présenté au chapitre 4. Cette API est utilisée pour développer les objets métier de l'application YAPS Pet Store.

/// JSR

JSR, ou *Java Specification Requests*, est un système normalisé ayant pour but de faire évoluer la plate-forme Java en donnant la possibilité à la communauté de créer de nouvelles spécifications. Vous pouvez retrouver toutes les spécifications sur le site du JCP.

► <http://www.jcp.org>

Bien que cette plate-forme soit prédite à un bel avenir, ses promesses ne sont pas toujours honorées. Les systèmes délivrés sont souvent trop lents et compliqués, et le temps de développement est, quant à lui, fréquemment disproportionné par rapport à la complexité des demandes des utilisateurs.

Heureusement, au deuxième trimestre 2006, Java EE 5 est venu simplifier la précédente version (J2EE 1.4). S'appuyant sur la nouvelle moulture du langage Java et s'inspirant de frameworks Open Source, certains composants de la version 5 de Java EE ont été totalement réécrits dans le but de simplifier la plate-forme.

La figure 2-3 décrit les différents conteneurs spécifiés dans Java EE 5 ainsi que les spécifications qui peuvent y être employées. Les paragraphes suivants nous donnent un bref descriptif des spécifications utilisées pour le développement de l'application YAPS Pet Store. Certaines ont vu le jour avec la version 5 de Java EE, d'autres ont été complètement remaniées pour simplifier le travail des développeurs.

JPA 1.0

Depuis les débuts de J2EE, le modèle de persistance ne cesse d'évoluer et de s'engluier de version en version. Les entity beans 2.0 ont été complètement réarchitecturés pour laisser place aux entity beans 2.1. Bien que cette évolution ait apporté beaucoup d'améliorations, ce modèle de composants persistants continuait à faire des détracteurs parmi la communauté. Ce mécontentement a donné naissance à une nouvelle spécification (JDO, *Java Data Object*) ainsi qu'à différents outils de mapping objet/relationnel payants ou libres (TopLink, Hibernate...).

Java EE 5, avec son lot de nouveautés, nous apporte un nouveau modèle de persistance : JPA (*Java Persistence API*). Fortement inspirés par des outils Open Source tels qu'Hibernate ou par JDO, le mapping objet/relationnel et le langage de requête sont totalement différents de l'ancêtre entity bean 2.1. JPA, ou JSR-220, réconcilie ainsi les utilisateurs de la plate-forme Java EE avec les composants persistants.

Java Persistent API s'appuie sur JDBC (*Java Data Base Connectivity*) pour communiquer avec la base de données. En revanche, grâce à l'abstraction apportée par JPA, nous n'aurons nul besoin d'utiliser directement JDBC dans le code Java.

JMS 1.1

Une des manières d'avoir des traitements asynchrones en Java EE, consiste en l'utilisation d'un MOM (*Message-Oriented Middleware*), c'est-à-dire un système basé sur l'échange de messages. En utilisant JMS (*Java*

Message Service), un client produit un message et le publie dans une file d'attente. Ainsi, la communication des événements ou des données se fait de façon asynchrone : ni le client ni l'EJB ne dépendent de la réponse directe de l'autre et leurs traitements ne sont donc pas figés durant l'attente d'une réponse.

EJB 3.0

Les *Entreprise Java Beans*, ou EJB, sont des composants serveurs qui respectent les spécifications d'un modèle édité par Sun. Ces spécifications définissent une architecture, un environnement d'exécution (un conteneur) et un ensemble d'API. Le respect de ces spécifications permet d'utiliser les EJB indépendamment du conteneur dans lequel ils s'exécutent. Ce dernier fournit un ensemble de fonctionnalités comme la gestion du cycle de vie de l'EJB, la sécurité, l'accès concurrent et les transactions. Le but des EJB est de faciliter la création d'applications distribuées pour les entreprises.

Pièce maîtresse de l'architecture Java EE, les EJB 3 apportent des modifications notables dans le mode de développement et intègrent de nombreuses nouveautés, notamment en ce qui concerne la persistance. Le passage des EJB 2.1 en 3.0 apporte une simplification radicale en supprimant les descripteurs de déploiement, les appels JNDI, etc., et laisse place aux annotations.

Il existe deux grandes familles d'EJB : entité et session. Les EJB sessions sont différenciés entre EJB sans état, avec état ou s'exécutant en mode asynchrone.

EJB stateless

Les EJB sans état, ou *stateless session beans*, ne fonctionnent que de façon éphémère. Une fois qu'un client a demandé et reçu une fonctionnalité du composant, l'interaction avec ce composant prend fin, ne laissant aucune trace de ce qui s'est passé. Ils n'ont aucune connaissance du client ou d'un semblant de contexte concernant l'enchaînement des requêtes : ils sont donc parfaits pour une utilisation unique. Ils n'ont pas d'état, c'est-à-dire qu'on ne peut pas manipuler leurs attributs en étant sûr de leur valeur.

L'utilisation standard d'un EJB sans état réside dans le fait qu'une application cliente le contacte en lui transmettant des paramètres. L'EJB accède alors à une base de données, effectue plusieurs traitements, appelle d'autres EJB, puis retransmet un résultat au client. Lorsque la communication s'achève, le bean ne conserve aucun souvenir de l'interaction. Avec ce comportement, plusieurs clients distincts peuvent accéder simultanément à un même stateless session bean.

À LIRE JMS

- 📖 Richard Monson-Haefel, *Java Message Service*, O'Reilly, 2002
- 📖 Eric Bruno, *Java Messaging*, Charles River Media, 2005

⚡ Serveur d'applications

Un serveur d'applications héberge les applications destinées à être utilisées dans un réseau distribué. Il est doté de services transactionnels entre composants, d'équilibrage de charge ou encore de tolérance aux pannes.

À LIRE EJB

- 📖 Laboratoire Supinfo, *EJB 3 : des concepts à l'écriture du code*, Dunod, 2006
- 📖 Rima Patel Sriganesh, Gerald Brose, Micah Silverman, *Mastering Enterprise JavaBeans*, Wiley, 2006

Le chapitre 5 présente les stateless session beans. Nous les utiliserons pour développer les composants métiers.

EJB Avec ou sans état ?

Un EJB stateless est utile pour calculer $\cos(x)$, convertir des euros en dollars, supprimer tous les bons de commandes passés il y a cinq ans ou obtenir le cours d'une action.

Un EJB stateful sert à stocker des articles achetés en ligne dans un panier électronique ou à commander plusieurs billets de train.

Le chapitre 8 présente les stateful session beans qui seront utilisés pour développer le panier électronique de l'application YAPS Pet Store.

L'API JMS et les message-driven beans sont décrits au chapitre 10. Les traitements asynchrones de l'application utilisent ces deux technologies.

Exemple d'EJB stateless

```
@Stateless
public class MonBean {
    // Le code métier
    public void maMethode() {
        return;
    }
}
```

EJB stateful

Par opposition au composant sans état, les *stateful session beans* associent les requêtes à un client spécifique, unissant client et EJB dans une relation un-un. Ce type de composant fournit ainsi un ensemble de traitements via des méthodes, mais il a aussi la possibilité de conserver des données entre les différents appels de méthodes d'un même client. Une instance particulière est donc dédiée à chaque client, qui sollicite ses services et ce, tout au long du dialogue.

Les données conservées par le bean sont stockées dans les variables d'instances. Les données sont donc conservées en mémoire. Généralement, les méthodes proposées par le composant permettent de consulter et de mettre à jour ces données.

Exemple d'EJB stateful

```
@Stateful
public class MonBean {
    // Attribut conservant sa valeur
    private String monAttribut;
    // Le code métier
    public void maMethode() {
        return;
    }
}
```

Message-driven bean

Les précédents types d'Enterprise Java Beans offrent des services de manière synchrone. Le client émet une requête, puis attend que l'EJB lui envoie un résultat.

Pour les *message-driven beans* (MDB), le comportement est complètement différent. Les clients n'appellent pas directement des méthodes mais utilisent JMS pour produire un message et le publier dans une file d'attente. À l'autre bout, le MDB est à l'écoute de cette file d'attente et se « réveille » à l'arrivée d'un message. Il extrait ce dernier de la file d'attente, en récupère le contenu puis exécute un traitement. Le client

n'a donc pas besoin de figer son exécution durant le traitement du MDB. Le traitement est asynchrone.

Exemple de MDB

```
@MessageDriven
public class MonMDB implements MessageListener {

    public void onMessage (Message msg) {
        // Traiter le message
    }
}
```

Entity

Les stateful session beans sont détruits lorsque la session du client se termine. Ils ne peuvent donc pas être utilisés pour stocker de façon permanente les informations de l'application. Les EJB entités peuvent répondre à ce besoin puisqu'ils sont persistants. En effet, leur état est sauvegardé sur un support de stockage externe, comme une base de données. Les entités représentent des données, ou plus exactement des objets métier, qui perdurent après la fin d'une session. Ils existent souvent sous la forme d'enregistrements uniques dans une base de données.

Depuis Java EE 5 et l'arrivée de JPA, on a plutôt tendance à parler d'entité (*entity*) que de bean entité (*entity bean*). En effet, si les entity beans 2.1 ont un modèle de développement assez lourd et compliqué, les entités sont, eux, de simples classes Java (Pojo) et peuvent même être utilisés en dehors de Java Enterprise Edition, c'est-à-dire dans une application Java standard. Il faut juste se faire à l'idée qu'un entity bean est devenu une simple classe Java (*lightweight*) et non un composant lourd (*heavyweight*) et complexe à développer.

Exemple d'entité d'adresse

```
@Entity
@Table(name = "t_adresse")
public class Adresse {

    @Id @GeneratedValue
    private Long identifiant;
    private String rue;
}
```

Le conteneur d'EJB

Souvent aussi appelé à tort *serveur d'applications*, le conteneur d'EJB a la responsabilité de créer de nouvelles instances d'EJB et de gérer leur cycle de vie. Il est l'intermédiaire entre l'EJB et le serveur d'applications. Il

≡ Pojo

Pojo est un acronyme qui signifie *Plain Old Java Object*. Ce terme est principalement utilisé pour faire référence à la simplicité d'utilisation d'un objet Java en comparaison avec la lourdeur d'utilisation d'un composant EJB.

ANNOTATIONS Java EE

Vous avez peut-être remarqué, dans les extraits de code précédents, l'utilisation des annotations Java dans le monde Java EE : `@Entity`, `@MessageDriven`, `@Stateless`, `@Stateful`. Comme nous le verrons dans les chapitres suivants, il en existe bien plus encore.

⚡ HTTP

HTTP, ou *Hypertext Transfer Protocol*, est un protocole de transfert de pages HTML sur le Web. Sa fonction première est d'établir la connexion avec un serveur, qui contient la page que l'on veut voir afficher, et de rapatrier cette page sur le poste de l'internaute.

À LIRE **Servlet et JSP**

- 📖 Jean-Luc Déléage, *JSP et Servlets efficaces*, Dunod, 2005
 - 📖 François-Xavier Sennesal, *JSP avec Eclipse et Tomcat*, ENI, 2007
 - 📖 Anne Tasso, Sébastien Ermacore, *Initiation à JSP*, Eyrolles, 2004
-

fournit des services tels que le transactionnel, la sécurité, la concurrence, la distribution, le service de nommage des EJB (JNDI) et l'exécution.

Les EJB interagissent avec le conteneur de plusieurs manières et peuvent exécuter des traitements déclenchés automatiquement par ce dernier. De même que si un EJB lève une exception, celle-ci est tout d'abord interceptée par le conteneur qui décidera d'effectuer telle ou telle action.

Servlet 2.5 et JSP 2.1

Les servlets sont des programmes Java fonctionnant côté serveur au même titre que les CGI et les langages de script tels que ASP ou PHP. Les servlets permettent donc de recevoir des requêtes HTTP, de les traiter et de fournir une réponse dynamique au client. Elles s'exécutent dans un conteneur utilisé pour établir le lien entre la servlet et le serveur web. Les servlets étant des programmes Java, elles peuvent utiliser toutes les API Java afin de communiquer avec des applications externes, se connecter à des bases de données, accéder aux entrées-sorties (fichiers, par exemple)...

Java Server Page, ou JSP, est une technologie basée sur Java qui permet aux développeurs de générer dynamiquement du code HTML, XML ou tout autre type de page web. Une page JSP (repérable par l'extension .jsp) aura un contenu pouvant être différent selon certains paramètres (des informations stockées dans une base de données, les préférences de l'utilisateur...) tandis qu'une page web « statique » (dont l'extension est .htm ou .html) affichera continuellement la même information.

Exemple de page JSP affichant la date du jour

```
<%@ page import="java.util.Date"%>
<html>
  <head>
    <title>JSP Affichant la date</title>
  </head>
  <body>
    <%! Date today = new Date();%>
    <br/>
    <center>La date est <%= today %></center>
  </body>
</html>
```

Une JSP est un autre moyen d'écrire une servlet. Lorsqu'un utilisateur appelle une page JSP, le serveur web crée un code source Java à partir du script JSP (c'est-à-dire qu'il constitue une servlet à partir du script JSP), le compile, puis l'exécute.

Langage d'expression

Le langage d'expression, ou *Expression language* (EL), permet aux JSP d'accéder aux objets Java, de manipuler des collections ou d'exécuter des actions JSF. Une expression est de la forme suivante :

```
${expression}
```

Exemple de page JSP utilisant le langage d'expression

```
<html>
<body>
  <c:if test="${bean.attr < 3}" >
    <center>La date est ${bean.today}</center>
  </c:if>
</body>
</html>
```

JSTL 1.2

JSTL est le sigle de *JSP Standard Tag Library*. C'est un ensemble de balises personnalisées (*Custom Tag*), développées sous la JSR 052 facilitant la séparation des rôles entre le développeur Java et le concepteur de pages web. L'avantage de ces balises est de déporter le code Java contenu dans la JSP dans des classes dédiées. Ensuite, il suffit de les utiliser dans le code source de la JSP en utilisant des balises particulières, tout comme vous le feriez avec des balises HTML classiques.

Les bibliothèques de balises (*Taglibs*) ou balises personnalisés (*Custom Tag*) permettent de définir ses propres balises basées sur XML, de les regrouper dans une bibliothèque et de les réutiliser dans des JSP. C'est une extension de la technologie JSP apparue à partir de la version 1.1 des spécifications.

Exemple d'utilisation d'une balise choose dans une page JSP

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
  <c:choose>
    <c:when test="${empty sessionScope.cart}">
      Le panier est vide
    </c:when>
    <c:otherwise>
      Le panier contient des articles
    </c:otherwise>
  </c:choose>
</html>
```

À LIRE JSF et Struts

- 📖 Bill Dudley, Jonathan Lehr, Bill Willis, LeRoy Mattingly, *Mastering JavaServer Faces*, 2004, Wiley
 - 📖 Jean-Charles Félicité, *Développement Java sous Struts*, ENI, 2006
 - 📖 Jean-Charles Félicité, *Struts 1.2 : les bases pour un développement Open Source Java*, ENI, 2006
 - 📖 Vic Cekvenich, Wolfgang Gehner, *Struts : les bonnes pratiques pour des développements web réussis*, Dunod, 2005
- <http://struts.apache.org/>
-

Le chapitre 7 se concentre sur le développement web de l'application YAPS Pet Store. Chaque API y est expliquée ainsi que la manière de les assembler pour obtenir un site web.

JAVAMAIL Les protocoles de messagerie

Le protocole SMTP (*Simple Mail Transfer Protocol*) est le protocole standard de messagerie. Le protocole POP (*Post Office Protocol*) permet d'aller récupérer son courrier sur un serveur distant et Imap (*Internet Message Access Protocol*) est une alternative à POP offrant plus de possibilités (comme la gestion d'accès simultanés, de plusieurs boîtes aux lettres...).

≡ XSD

XSD, ou *XML Schema Description*, est un langage de description de format de document XML permettant de définir la structure d'un document XML. XSD est communément appelé *grammaire*.

JSF 1.2

Entre les servlets et les JSP, il manquait un framework pour aiguiller, de manière simple, un événement utilisateur vers une action serveur. Des outils libres comme Struts sont venus aider le développeur en décorrélant la couche présentation de la couche métier. Mais aucune spécification n'existait jusqu'à l'apparition de JSF. *Java Server Faces* est venu combler ce vide en facilitant la conception d'interfaces graphiques web, en gérant automatiquement l'état HTTP ainsi que les événements entre client et serveur.

JSF établit une norme dont le rôle est de fournir aux développeurs une large palette d'outils leur permettant d'implémenter des applications web en respectant un modèle bien précis.

Le conteneur de servlet

Le cycle de vie d'une servlet, donc d'une JSP, est assuré par le moteur de servlet (aussi appelé *conteneur de servlet* ou *conteneur web*). Celui-ci est responsable de fournir la requête HTTP à la servlet, de l'exécuter et de renvoyer la réponse. C'est le « moteur » de toute application web simple, c'est-à-dire ne mettant pas en jeu d'EJB.

JavaMail 1.4

JavaMail est l'API standard de gestion de courriers électroniques de Java EE. Elle permet d'envoyer et de recevoir du courrier électronique et de manipuler les messages (en-tête, sujet, corps, pièces jointes...). JavaMail n'est pas un serveur de courrier en tant que tel, mais plutôt un outil pour interagir avec ce type de serveur. Il peut être vu comme un type de client de messagerie au même titre que Outlook, Lotus, Eudora, etc. Pour envoyer ou recevoir des messages, JavaMail utilise différents protocoles comme SMTP, Imap ou POP.

JAXB 2.0

JAXB est l'acronyme de *Java Architecture for XML Binding*. Cette API permet de générer des classes Java à partir de schémas XML (XSD) et inversement. Autrement dit, il permet de convertir les fichiers XSD en classes Java. Il est ensuite possible de manipuler le document XML au travers de ces classes.

Une fois de plus, les annotations de Java 5 sont venues simplifier l'utilisation de l'API JAXB. En annotant un Pojo (*Plain Old Java Object*), on peut ensuite obtenir ses attributs au format XML.

Exemple d'annotations JAXB

```
@XmlRootElement
public class Adresse {

    @XmlID
    private Long identifiant;
    private String rue;
}
```

Grâce aux annotations situées dans la classe, JAXB pourra générer automatiquement la structure XML suivante :

XML généré par ces annotations

```
<adresse>
  <identifiant> </identifiant>
  <rue> </rue>
</adresse>
```

Services web

Comment faire dialoguer des logiciels écrits dans des langages de programmation différents et fonctionnant sur des systèmes d'exploitation divers et variés ? La réponse est simple : en utilisant des services web. Les services web permettent cette interopérabilité en s'appuyant sur un ensemble de protocoles répandus comme HTTP. Cette communication est basée sur le principe de demandes et réponses, effectuées via des messages XML.

Les services web sont décrits par des documents WSDL (*Web Service Description Language*), qui précisent les méthodes pouvant être invoquées, leurs signatures et les points d'accès de service (URL, port). Les services web sont accessibles via Soap, la requête et les réponses sont des messages XML transportés sur HTTP.

Exemple de service web

```
@WebService
public class MonWebService {
    // Le code métier
    public void maMethode() {
        return;
    }
}
```

Blueprints

Parallèlement à la plate-forme Java EE, Sun propose gratuitement des documents pour faciliter les développements Java : les *Blueprints*. Ces

À LIRE Services web

- 📖 Annick Fron, *Architectures réparties en Java : RMI, CORBA, JMS, sockets, SOAP, services web*, Dunod, 2007
- 📖 Hubert Kadima, Valérie Monfort, *Les Web Services - Techniques, démarches et outils*, Dunod, 2003
- 📖 Libero Maesano, Christian Bernard, Xavier Le Galles, *Services Web avec J2EE et .NET*, Eyrolles, 2003
- 📖 Steve Graham, Doug Davis, Simeon Simeonov, Glen Daniels, *Building Web Services with Java: Making Sense of XML, Soap, WSDL, and UDDI*, Sams, 2004

Le chapitre 9 présente les services web ainsi que les technologies qui y sont rattachées. Les services web sont utilisés par l'application YAPS Pet Store pour communiquer avec les partenaires externes.

/// Soap

Simple Object Access Protocol est un protocole standard destiné aux services web. Lancé par IBM et Microsoft, il permet d'utiliser des applications invoquées à distance par Internet.

ARCHITECTURE **Pet Store**

Les Blueprints de Sun se trouvent à l'adresse suivante :

- ▶ <http://java.sun.com/reference/blueprints/>
- En ce qui concerne le Pet Store, vous pouvez consulter les adresses suivantes :
- ▶ <http://blueprints.dev.java.net/petstore/>
- ▶ <http://java.sun.com/developer/releases/petstore/>

REMARQUE **Les autres Pet Store**

Le Pet Store de Sun a été source d'inspiration pour d'autres technologies ou frameworks. Ci-dessous une liste non exhaustive de ces Pet Store :

- PetShop : utilisation du framework .NET de Microsoft ;
- xPetStore : utilisation des tags xDoclet ;
- Flash PetStore : version de Macromedia utilisant la technologie Flash ;
- Spring PetStore : utilisation du framework Spring ;
- openMDX PetStore : plate-forme Open Source MDA.

Figure 2-4

Page d'accueil du Java Pet Store de Sun

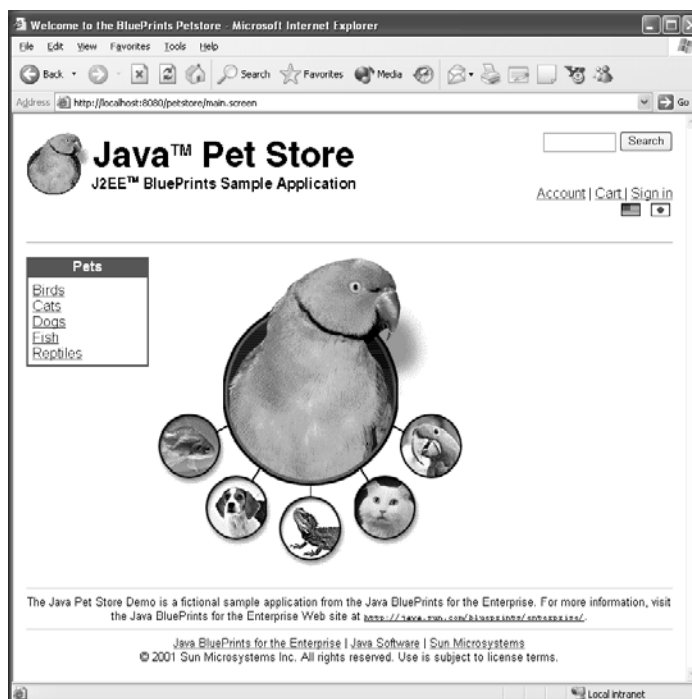
derniers contiennent des tutoriaux, des design patterns, des exemples de code, des conseils et des FAQs.

Il existe plusieurs types de Blueprints. Sous le succès des services web, Sun a développé en 2004 un Blueprint baptisé Adventure Builder. Cette application vous permet de personnaliser un séjour pour vos vacances, en utilisant principalement des services web.

Concernant Java EE (ou J2EE à l'époque), Sun créa le Java Pet Store.

Java Pet Store

Java Pet Store est une application Java EE que Sun a créé pour son programme de Blueprints. C'est un site web marchand où l'on peut choisir des animaux domestiques, les ajouter dans un panier, puis payer électroniquement. Ce Blueprint a permis de documenter les meilleures pratiques (code Java, design pattern, architecture) pour développer une application Java EE.



Le Java Pet Store est aussitôt devenu un standard de facto, puisque les constructeurs de serveur d'applications l'ont utilisé pour démontrer la compatibilité de leur produit avec les spécifications Java EE. En effet, Oracle fut le premier à l'utiliser pour ses tests de montée en charge. Bien que Java Pet Store ait été développé à des fins éducatives, Oracle déclara que cette application fonctionnait deux fois plus rapidement sur son ser-

veur d'applications que sur ceux de BEA ou IBM. La communauté s'enflamma et tous les vendeurs commencèrent à utiliser le Java Pet Store pour démontrer leurs meilleures performances.

Cette anecdote contribua à augmenter la popularité de ce Blueprint, qui rentra très vite dans le langage commun. Tout le monde commença à l'utiliser pour illustrer une nouvelle technologie, une nouvelle idée ou implémentation.

L'étude de cas de cet ouvrage s'inspire de ce site de commerce électronique.

Les design patterns

Dans son livre *A Pattern Language* édité en 1977, l'architecte en bâtiment Christopher Alexander introduit le terme de *pattern* (patron) : « chaque patron décrit un problème qui se produit de manière récurrente dans notre environnement ». Si ce livre est dédié à une autre profession que celle de l'architecture informatique, il faudra attendre le livre du Gang of Four (GoF) en 1994 pour adapter ces idées au monde de l'orienté objet.

Il ne faut pas confondre ces patrons avec des briques logicielles (un pattern dépend de son environnement), des règles (un pattern ne s'applique pas mécaniquement) ou des méthodes (ne guide pas la prise de décision). Mais plutôt les voir comme une solution de conception à un problème récurrent.

Viendront alors, bien plus tard, deux livres s'inspirant du GoF mais dédié à la plate-forme J2EE : *EJB Design Pattern* et *Core J2EE Patterns*. Ces trois ouvrages ont créé un vocabulaire commun entre les développeurs, concepteurs et architectes.

Ce livre utilisera plusieurs design patterns pour concevoir l'application YAPS Pet Store.

UML 2

UML (*Unified Modeling Language* ou langage de modélisation unifié), est né de la fusion des trois méthodes qui ont le plus influencé la modélisation objet au milieu des années 1990 : OMT, Booch et OOSE. Issu d'un travail d'experts reconnus (James Rumbaugh, Grady Booch et Ivar Jacobson), UML est le résultat d'un large consensus qui est vite devenu un standard incontournable. Fin 1997, ce langage est devenu une norme OMG (*Object Management Group*).

UML est un langage de modélisation objet et non une démarche d'analyse. Il représente des concepts abstraits de manière graphique. UML est donc un langage universel et visuel qui permet d'exprimer et d'élaborer des modèles

GoF

Le *Gang of Four* désigne les quatre auteurs du livre *Design Pattern*, c'est-à-dire Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides

Les anti-patterns

Les anti-patterns sont des erreurs courantes de conception de logiciels. Leur nom vient du fait que ces erreurs sont apparues dès les phases de conception du logiciel, notamment par l'absence ou la mauvaise utilisation de design pattern.

À LIRE Design pattern

- 📖 Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Pattern*, Addison-Wesley, 1995
- 📖 Floyd Marinescu, *EJB Design Patterns*, Wiley, 2002
- 📖 Deepak Alur, Dan Malks, John Crupi, *Core J2EE Patterns*, Prentice Hall, 2003
- 📖 Laurent Debrauwer, *Design Patterns – Les 23 modèles de conception : descriptions et solutions illustrées en UML 2 et Java*, ENI, 2007
- 📖 Steven-John Metsker, William C. Wake, *Les Design Patterns en Java*, Campus Press, 2006

OMG

L'objectif de l'*Object Management Group* est de standardiser et de promouvoir le modèle objet sous toutes ses formes. L'OMG est notamment à la base des spécifications UML, MOF, CORBA, IDL et MDA.

APPROFONDIR UML

- 📖 Pascal Roques, *UML 2 – Modéliser une application web*, Eyrolles, 2007
- 📖 Pascal Roques, Franck Vallée, *UML 2 en action : de l'analyse des besoins à la conception*, Eyrolles, 2007
- 📖 Xavier Blanc, Isabelle Mounier, *UML 2 pour les développeurs – Cours avec exercices corrigés*, Eyrolles, 2006
- 📖 Jim Arlow, Ila Neustadt, *UML2 and the Unified Process*, Addison-Wesley, 2005
- <http://www.uml.org/>

Architecture

L'architecture spécifie la structure d'un système. On parle d'architecture fonctionnelle pour définir les services du système, d'architecture technique pour les composants techniques utilisés et d'architecture applicative pour décrire le découpage en sous-systèmes.

Architecture Couches ou tiers

Lorsqu'on parle d'architecture en couches, on utilise souvent le terme anglais *tiers*. Ce terme signifie couche et non le tiers mathématique (1/3). On entend par conséquent les architectes parler d'architecture quatre tiers ou cinq tiers. Il ne faut pas comprendre par là que l'architecture est en 4/3 ou 5/3 mais bien qu'elle est découpée en 4 ou 5 couches.

UML Paquetages et sous-systèmes

Un paquetage (*package* en anglais) est un mécanisme destiné à regrouper des éléments comme des classes, des cas d'utilisation, voire d'autres paquetages. Le terme sous-système (*subsystem*) indique que le paquetage représente une partie indépendante du système. Ci-après la représentation graphique UML d'un paquetage et d'un sous-système.

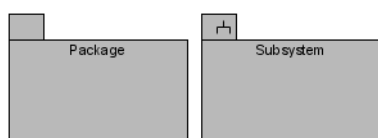


Figure 2-5 Paquetage et sous-système

objet, indépendamment de tout langage de programmation. Comme UML n'impose pas de méthodes de travail particulières, il peut être intégré à n'importe quel processus de développement logiciel de manière transparente.

UML 2 introduit quatre nouveaux diagrammes (paquetages, structures composites, global d'interaction et de temps) qui viennent enrichir les neuf initiaux (classes, objets, composants, déploiement, cas d'utilisation, états-transitions, activités, séquence et communication). La plupart de ces diagrammes seront utilisés tout au long des chapitres.

Architecture de l'application

L'application YAPS Pet Store va donc utiliser toutes les technologies énoncées ci-dessus. Comme nous l'avons vu au chapitre précédent, *Présentation de l'étude de cas*, on accède à l'application par des navigateurs (client léger utilisé par les internautes) et par les clients riches (développés en Swing) déployés sur les postes des employés. Ces interfaces graphiques accèdent elles-mêmes à un serveur qui va effectuer les traitements métier puis stocker les données dans une base.

Pour ce faire, l'application YAPS Pet Store est découpée en plusieurs couches.

L'architecture en trois couches

On peut donc dire que l'architecture logique de l'application YAPS Pet Store est découpée en trois couches (ou trois niveaux). L'architecture en trois couches est le modèle le plus général des architectures multicouches. Ces couches sont :

- **présentation des données** : affichage sur le poste de travail des données du système et interaction avec l'utilisateur ;
- **traitements métier** : ensemble des règles métiers de l'application ;
- **accès aux données** : manipulation et conservation des données.

Ci-après un diagramme de paquetages UML représentant ces trois couches. Chaque sous-système contient les technologies Java EE 5 utilisées dans l'application.

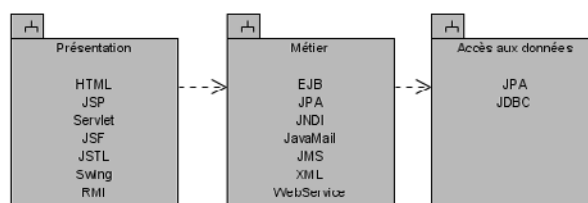


Figure 2-6
Architecture Java EE
en trois couches

Architecture applicative

Le précédent modèle en trois couches peut être affiné pour être plus fidèle à l'architecture finale de l'application YAPS Pet Store. Ci-après un diagramme de paquetage décrivant les couches de l'application :

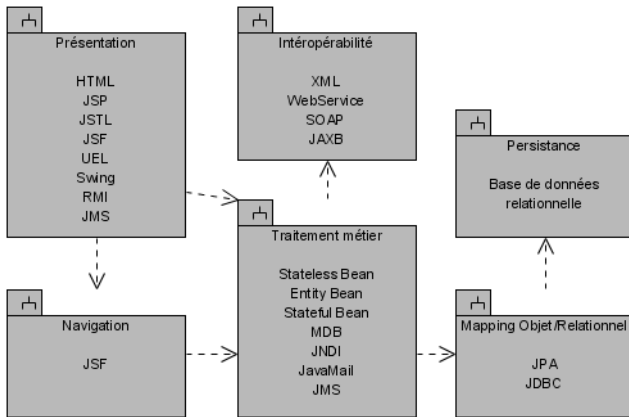


Figure 2-7
Couches de l'application YAPS Pet Store

Couche de présentation

La couche de présentation est la partie visible de l'application qui permet à un utilisateur d'interagir avec le système. Elle relaie les requêtes de l'utilisateur à destination de la couche métier, et en retour lui présente les résultats renvoyés par les traitements. On parle alors d'interface homme-machine (IHM) et aucun traitement n'est implémenté dans cette couche. L'application YAPS Pet Store possède deux types d'interfaces homme-machine : un client léger et un client riche.

Le client léger, utilisé par l'internaute, est décrit en langage HTML puis interprété par le navigateur. Dans le cas du YAPS Pet Store, nous ne développerons pas directement l'interface en HTML, mais plutôt à l'aide de JSP et de tags JSF et JSTL ainsi que du langage d'expression. Les appels à la couche métier sont délégués à la couche navigation.

Le client riche, lui, est développé en Swing et utilise le protocole RMI (*Remote Method Invocation*) pour communiquer avec la couche métier. Pour afficher les événements asynchrones reçus de l'application, cette couche utilise également JMS.

Couche de navigation

Cette couche, uniquement utilisée par le client léger, prend en charge la logique de navigation. De ce fait, elle gère l'enchaînement des JSP ainsi que les appels aux traitements métier. Cette couche est mise en œuvre par la technologie JSF.

JMS L'application Swing

L'expression des besoins nous signale que les employés veulent être avertis lorsqu'un bon de commande contient des reptiles. Pour ce faire, l'application écoutera sur une file d'attente JMS et sera alertée à chaque fois qu'un reptile est vendu.

/// CRUD

CRUD est un terme communément utilisé pour l'accès aux bases de données. Il signifie en anglais *Create, Retrieve, Update and Delete*, c'est-à-dire création, lecture, mise à jour et suppression de données.

/// Les bases de données objets

Les bases de données objets, comme leur nom l'indique, organisent les données sous forme d'objets et non sous forme de tables (lignes et colonnes). Bien que certains grands acteurs du monde relationnel aient des implémentations objets, les bases objets n'ont jamais vraiment percé sur le marché.

/// JAX-WS

JAX-WS est la nouvelle appellation de JAX-RPC (*Java API for XML Based RPC*) qui permet de développer très simplement des services web.

Couche de traitement métier

La couche de traitement métier correspond à la partie fonctionnelle ou métier de l'application. Elle implémente la logique et les règles de gestion permettant de répondre aux requêtes de la couche présentation.

Pour fournir ces services, elle s'appuie, le cas échéant, sur les données du système, accessibles au travers des services de la couche inférieure, c'est-à-dire la couche de données. En retour, elle renvoie à la couche présentation les résultats qu'elle a calculés.

En pratique, on trouve au niveau de la couche métier :

- des **entités** dont la persistance est assurée par la couche de mapping ;
- des **stateless beans** qui proposent des méthodes pour manipuler les entités (CRUD) ;
- des **message-driven beans** qui assurent les traitements asynchrones ;
- les **API JNDI**, pour accéder au service de nommage, et **JavaMail**, pour envoyer des e-mails aux clients.

Les appels vers les systèmes externes (BarkBank et PetEx) sont orchestrés par la couche métier mais délégués à la couche d'interopérabilité.

Couche de mapping objet/relationnel

La couche de mapping objet/relationnel transforme la représentation physique des données en une représentation objet et inversement. Ce mécanisme est assuré par JPA qui utilise le protocole JDBC pour exécuter les appels SQL. Cette couche n'est utilisée que parce que notre base de données est relationnelle. En effet, si la persistance était faite de manière native en objet, ce mapping n'aurait pas lieu d'être.

Couche de persistance

Cette couche contient les données sauvegardées physiquement sur disque, c'est-à-dire la base de données. En Java, le protocole d'accès aux données est JDBC.

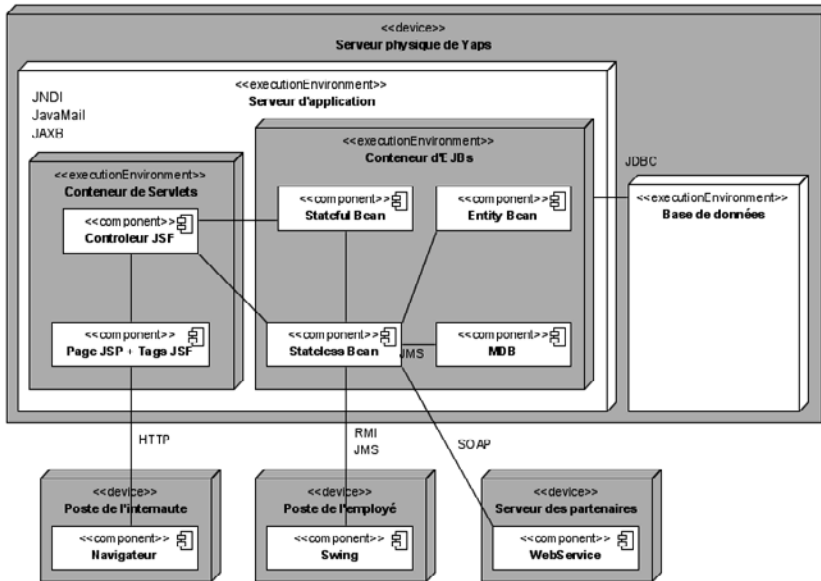
Couche d'interopérabilité

Pour dialoguer avec ses partenaires BarkBank et PetEx, l'application utilise une couche d'interopérabilité. Basée sur les technologies JAX-WS, elle accède à des services web distants via le protocole HTTP.

Architecture technique

Les couches que nous venons de décrire sont avant tout logiques et servent à décrire la conception de l'application. Nous allons maintenant

projeter cette architecture sur un ou plusieurs emplacements physiques. Le diagramme de déploiement ci-après nous montre les machines physiques utilisées. Pour l'application à proprement parler, il n'y a dans notre cas qu'un seul serveur physique. Mais nous aurions pu subdiviser. Les autres machines correspondent aux postes des internautes, des employés et des partenaires externes.



Le serveur physique comporte une base de données et un serveur d'applications. Ce dernier est composé d'un conteneur de servlets et d'un conteneur d'EJB. À l'intérieur de chaque nœud (représenté sous forme de cube), on peut voir les composants qui sont déployés. Pour la partie web, on retrouve les JSP, JSF et JSTL, alors que le conteneur d'EJB héberge les stateless, stateful, MDB et entities. Les services web sont déployés sur les serveurs des partenaires.

En résumé

Ce chapitre nous a présenté les différents langages ainsi que la plateforme Java EE 5 avec lesquels sera développée l'application YAPS Pet Store. L'architecture en couches a été détaillée à l'aide de diagrammes de paquetages et de déploiement. Nous avons désormais défini et recensé les technologies et spécifications qui seront utilisées dans les différentes couches applicatives.

UML Le diagramme de déploiement

Le diagramme de déploiement montre la disposition physique des matériels qui composent le système et la répartition des composants sur ces matériels. Les ressources matérielles sont représentées sous forme de nœuds (les cubes) qui peuvent être liés entre eux à l'aide d'un support de communication. Le stéréotype `<<device>>` renforce le fait que le nœud est un serveur physique, alors que `<<executionEnvironment>>` est utilisé pour les logiciels tels qu'un serveur d'applications ou de base de données.

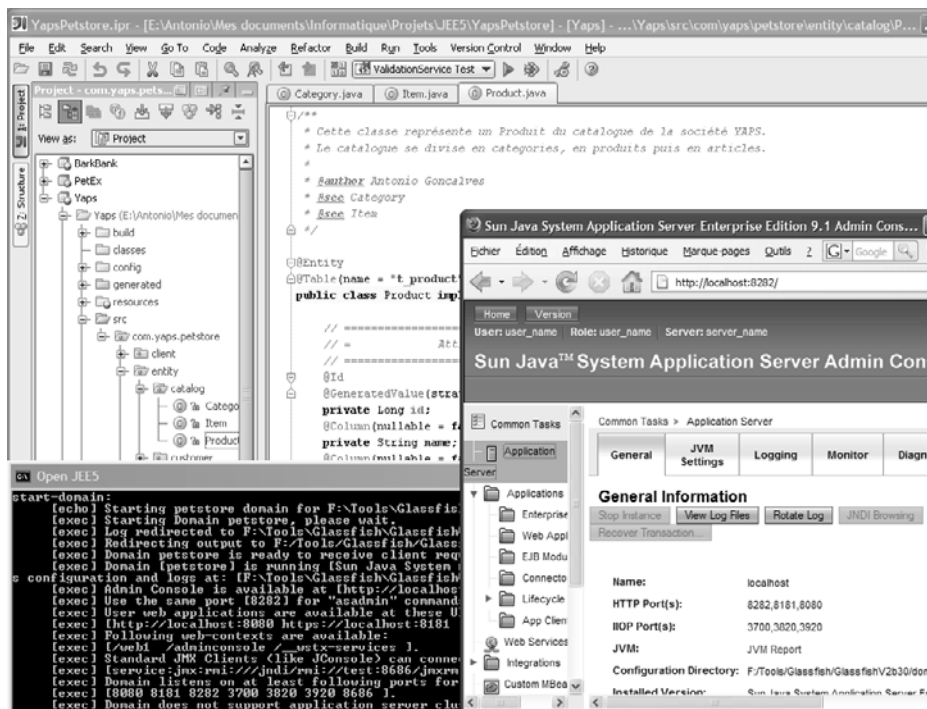
Figure 2-8
Diagramme de déploiement de l'application

PRÉCISION Machine physique

La répartition des composants sur différentes machines est une vision idéale de l'architecture. Pour simplifier le déploiement de la totalité de l'application, nous n'utiliserons qu'une seule machine qui hébergera le serveur d'applications, les services web ainsi que les interfaces homme-machine.

3

chapitre



Outils et installation

Les chapitres précédents ont permis de présenter les fonctionnalités, le contenu de l'application YAPS Pet Store ainsi que son architecture et les technologies utilisées. Avant de commencer les développements, ce chapitre nous présente les outils que nous allons utiliser, leur installation et leur configuration.

SOMMAIRE

- ▶ Outils utilisés pour le développement
- ▶ Installation de Ant et du JDK
- ▶ Installation et configuration du serveur GlassFish
- ▶ Administration de GlassFish
- ▶ Installation de Derby
- ▶ Création de la base de données Derby

MOTS-CLÉS

- ▶ JDK
- ▶ Ant
- ▶ GlassFish
- ▶ Derby
- ▶ DataSource
- ▶ Pool de connexions
- ▶ JMS

// Logiciel libre

Ce qui caractérise les logiciels libres (Open Source), c'est leur code source. En effet, celui-ci est visible, modifiable et librement redistribuable sous certaines conditions (licence).

TÉLÉCHARGER **JDK 1.5**

► <http://java.sun.com/javase/>

TÉLÉCHARGER **Ant 1.7**

► <http://ant.apache.org/>

Plusieurs outils seront utilisés pour développer notre étude de cas. Ils ont tous la particularité d'être gratuits et parfois même Open Source (logiciels libres). Il ne vous en coûtera donc rien de les installer et de les utiliser.

Outils utilisés pour le développement de l'application

JDK

Indispensable pour le développement et l'exécution de notre application, le Java Development Kit, communément appelé JDK, est le kit de développement proposé gratuitement par Sun. Il comprend plusieurs outils, parmi lesquels :

- javac : le compilateur Java ;
- java : un interpréteur d'applications (machine virtuelle) ;
- javadoc : un générateur de documentation ;
- jar : un outil de compression de classes Java.

Le JDK nous permettra de compiler et d'exécuter l'application, de même que d'autres outils tels que Ant.

Ant

Ant est au monde Java ce que Make est au monde du langage C : un outil incontournable pour automatiser des traitements répétitifs en mode batch (suppression de fichiers, compilation, compression de fichiers, etc.). Il est simple d'utilisation, bâti sur des technologies ouvertes (Java et XML), extensible et supporté par de nombreux logiciels. Cet outil est aujourd'hui plébiscité par l'ensemble des acteurs majeurs de la communauté Java et communément employé dans la majorité des réalisations d'entreprise

Ant sera utilisé pour automatiser la compilation, le packaging et le déploiement de l'application. Il nous permettra aussi d'insérer physiquement des données en base et d'administrer le serveur d'applications.

GlassFish

GlassFish est un serveur d'applications certifié Java EE 5. Son développement a été initié lorsque Sun a ouvert le code de son serveur d'applica-

tions pour le licencier en Open Source. Il utilise le moteur de persistance d'Oracle, TopLink Essentials.

GlassFish est constitué de :

- un serveur web dédié au service de fichiers, c'est-à-dire à des pages HTML statiques, images, vidéos, etc. ;
- un conteneur de servlets, hébergeant des applications composées de servlets et/ou JSP ;
- un conteneur d'EJB, pour la gestion des composants stateless, stateful, MDB et entités ;
- l'implémentation de l'API de persistance JPA d'Oracle (TopLink Essentials).

Comme nous le verrons, l'administration du serveur GlassFish se fait soit par interface web, soit par ligne de commande.

GlassFish hébergera l'application YAPS Pet Store ainsi que les services web des partenaires BarkBank et PetEx.

Derby

Anciennement appelée Cloudscape, cette base de données développée en Java a été donnée à la fondation Apache par IBM. De petite taille (2 Mo), cette base de données relationnelle est intégrée au serveur GlassFish. Nous utiliserons Derby pour stocker les données de l'application.

Environnement de développement

L'environnement minimal pour développer l'application se compose d'un JDK, de Ant et d'un simple éditeur de texte. Cependant, il est important d'avoir un outil intégré pour vous permettre d'accélérer vos développements (IDE ou Integrated Development Environment). Vous pouvez ainsi utiliser Eclipse, ou NetBeans, en ce qui concerne les outils Open Source, ou tout autre IDE de votre choix si vous en possédez la licence.

Pour ma part, j'utilise depuis longtemps IntelliJ IDEA de JetBrains (voir annexe E). À ce sujet, je remercie la société JetBrains de m'avoir offert une licence pour la version 7 de leur excellent produit.

Outil de modélisation UML

Si vous voulez dessiner des diagrammes UML, il existe plusieurs outils disponibles en Open Source (ArgoUML, StarUML, etc.) ou sous forme de plug-in pour Eclipse, NetBeans ou Idea.

SERVEUR D'APPLICATIONS **Compatibilité Java EE 5**

GlassFish n'est pas le seul serveur d'applications Open Source à supporter la totalité des spécifications Java EE 5. Le code que nous allons développer est portable à 99 % sur tout serveur d'applications. Retrouvez toutes les informations nécessaires pour migrer vers tel ou tel autre serveur d'applications à l'URL suivante :

► <http://www.antoniogoncalves.org>

TÉLÉCHARGER **GlassFish**

► <https://glassfish.dev.java.net/>

TÉLÉCHARGER **Derby**

► <http://db.apache.org/derby/>

OUTILS **Eclipse, NetBeans, IntelliJ IDEA**

Initialement lancé par IBM, Eclipse est un IDE Open Source extensible et polyvalent. Utilisé par beaucoup de développeurs Java, vous pourrez le télécharger à l'adresse suivante :

► <http://www.eclipse.org/>

NetBeans est le pendant chez Sun :

► <http://www.netbeans.org/>

IntelliJ IDEA est payant, mais vous pouvez utiliser une licence d'évaluation pour essayer le produit :

► <http://www.jetbrains.com/idea/>

UML Outils de modélisation

ArgoUML, StarUML et Poséidon sont des outils Open Source que vous pourrez retrouver respectivement aux adresses suivantes :

- ▶ <http://argouml.tigris.org/>
- ▶ <http://staruml.sourceforge.net>
- ▶ <http://www.gentleware.com>

Visual Paradigm possède une large panoplie d'outils de modélisation dont la Community Edition qui est gratuite :

- ▶ <http://www.visual-paradigm.com/>

TÉLÉCHARGER JDK 1.5

- ▶ http://java.sun.com/javase/downloads/index_jdk5.jsp

Les diagrammes que vous verrez dans ce livre ont été dessinés avec l'outil Visual Paradigm. Cet outil s'intègre aussi sous forme de plug-in dans IntelliJ IDEA, Eclipse, NetBeans et bien d'autres IDE. Je remercie la société Visual Paradigm de m'avoir offert une licence pour leur édition Smart Development Environment.

Installation des outils

Dans cette section, nous allons installer les trois principaux outils utilisés tout au long de l'ouvrage.

JDK 1.5

Téléchargez l'exécutable d'installation sur le site officiel de Sun. Le nom de cet exécutable diffère selon la plate-forme sur laquelle vous souhaitez l'installer. Par exemple :

- Windows : `jdk-1_5_0_15-windows-i586-p.exe` ;
- Linux : `jdk-1_5_0_15-linux-i586-rpm.bin` ;
- Solaris Sparc : `jdk-1_5_0_15-solaris-sparc.sh`.

L'installation se fait alors sans difficultés particulières. Un premier écran vous invite à accepter la licence du logiciel, puis le second les modules du JDK que vous souhaitez installer (figures 3-1 et 3-2). L'installation s'achève avec un écran vous informant de son bon déroulement.

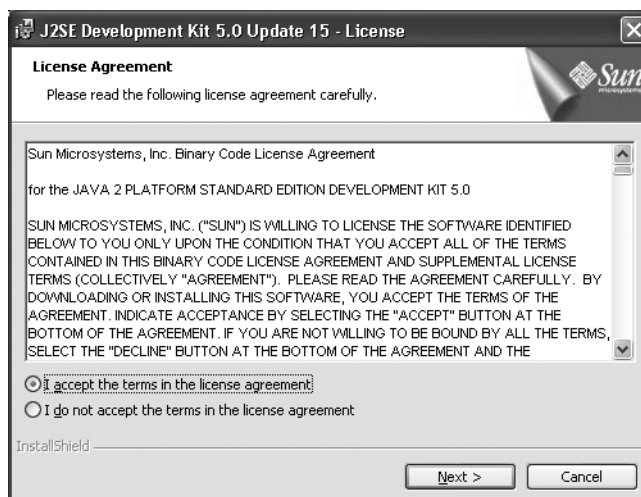


Figure 3-1

Acceptez la licence et cliquez sur Next.

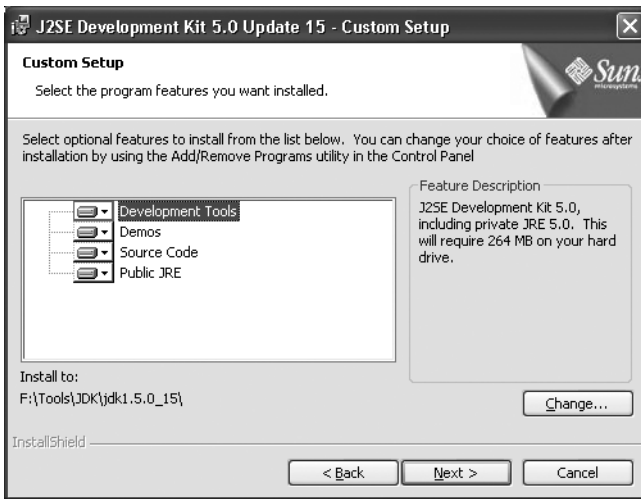


Figure 3-2
Sélectionnez les modules à installer
et cliquez sur Next.

L'installation terminée, il faut positionner la variable `JAVA_HOME` (par exemple, set `JAVA_HOME=F:\Tools\JDK\jdk1.5.0_15`) et rajouter le répertoire `bin` dans la variable système `PATH` (pour notre exemple `PATH=%JAVA_HOME%\bin;%PATH%`) à partir d'une fenêtre de commande. Vérifiez que l'interpréteur `java` est reconnu en tant que commande interne en tapant la commande `java -version` dans votre fenêtre de commande (figure 3-3).



Figure 3-3
Affichage de la version du JDK
dans une fenêtre de commande

RETOUR D'EXPÉRIENCE Des espaces dans les chemins

Java n'apprécie guère que vous installiez des outils ou des bibliothèques dans des répertoires contenant des espaces (par exemple, `c:\Program Files` ou `d:\Mes Outils`). Il faut alors rajouter des guillemets sur les chemins d'accès (par exemple, "`c:\Program Files`" ou "`d:\Mes Outils`"). Les applications Java ont fréquemment besoin de bibliothèques externes qui doivent être rajoutées dans la variable `Classpath`. On se retrouve alors avec des erreurs de type `ClassNotFoundException` car les chemins d'accès aux bibliothèques contiennent des espaces et que l'interpréteur Java n'arrive pas à retrouver les classes nécessaires. Pour éviter ce genre de problèmes et devoir rajouter des guillemets un peu partout, vous pouvez installer vos outils et bibliothèques dans des répertoires ne contenant pas d'espace (par exemple, `c:\Tools` ou `d:\MesOutils`).

TÉLÉCHARGER **Ant 1.7**

► <http://ant.apache.org/bindownload.cgi>

REMARQUE **Ant 1.6.5**

Les scripts ont été testés et fonctionnent correctement pour les versions 1.7 et 1.6.5 de Ant.

Figure 3–4

Affichage de la version de Ant
dans une fenêtre de commande

TÉLÉCHARGER **Les fichiers
admin.xml et build.xml**

Téléchargez le code source de l'application YAPS Pet Store. À la racine de l'archive, vous trouverez les deux fichiers `admin.xml` et `build.xml`.
► <http://www.antoniogoncalves.org>

TÉLÉCHARGER **GlassFish V2**

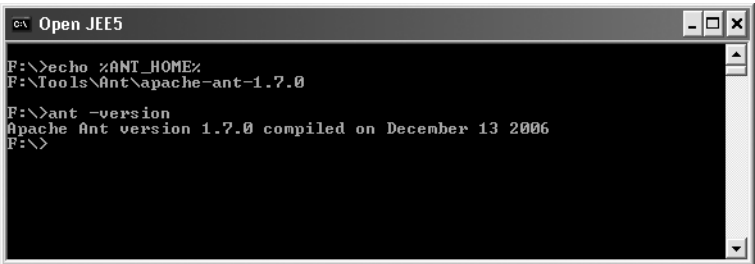
► <https://glassfish.dev.java.net/public/downloadsindex.html>

Ant 1.7

L'installation de Ant se limite à décompresser un fichier. Selon la plateforme et le mode de compression utilisés, vous téléchargerez le fichier suivant :

- `apache-ant-1.7.0-bin.tar.bz2` ;
- `apache-ant-1.7.0-bin.tar.gz` ;
- `apache-ant-1.7.0-bin.zip`.

Décompressez le fichier téléchargé dans un répertoire et positionnez la variable `ANT_HOME` (par exemple, `set ANT_HOME=F:\Tools\Ant\apache-ant-1.7.0`). Rajoutez le répertoire `%ANT_HOME%\bin` dans la variable système `PATH`. Vérifiez que l'interpréteur ant est reconnu en tant que commande interne en tapant la commande `ant -version` dans une fenêtre de commande (figure 3–4).



```
F:\>echo %ANT_HOME%
F:\Tools\Ant\apache-ant-1.7.0

F:\>ant -version
Apache Ant version 1.7.0 compiled on December 13 2006
F:\>
```

Une fois Ant installé, nous pourrons l'utiliser pour administrer et développer l'application YAPS Pet Store. Les différentes tâches Ant sont regroupées dans deux fichiers distincts :

- `admin.xml` : contient les tâches d'administration ;
- `build.xml` : les tâches pour le développement de l'application.

GlassFish V2

L'installation du serveur GlassFish nécessite plus d'opérations que les installations précédentes, mais reste tout de même très simple. Tout d'abord, téléchargez le fichier compressé correspondant à votre plateforme :

- Windows : `glassfish-installer-v2ur2-b04-windows.jar` ;
- Linux : `glassfish-installer-v2ur2-b04-linux.jar` ;
- Solaris Sparc : `glassfish-installer-v2ur2-b04-sunos.jar`.

Décompressez le fichier en tapant la commande suivante (pour Windows) :

```
java -Xmx256m -jar glassfish-installer-v2ur2-b04-windows.jar
```

Une fenêtre s'affiche contenant la licence d'utilisation du produit. Vous devez faire défiler l'écran et cliquer sur **Accept** (figure 3-5).

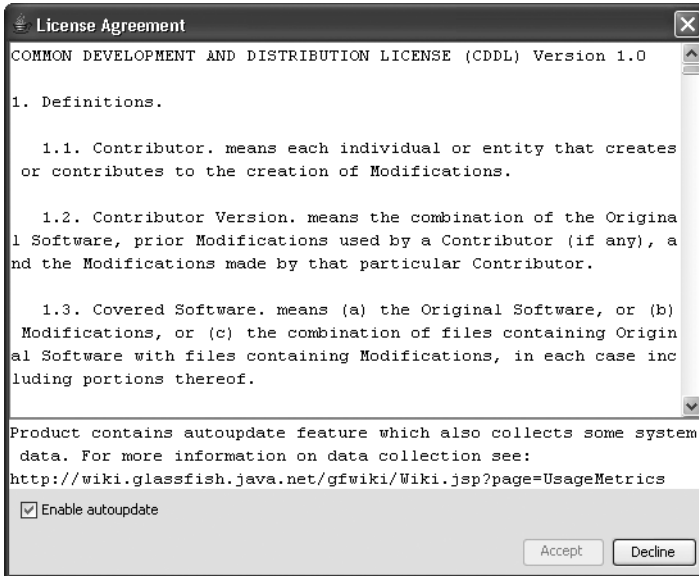


Figure 3-5
Affichage de la licence GlassFish

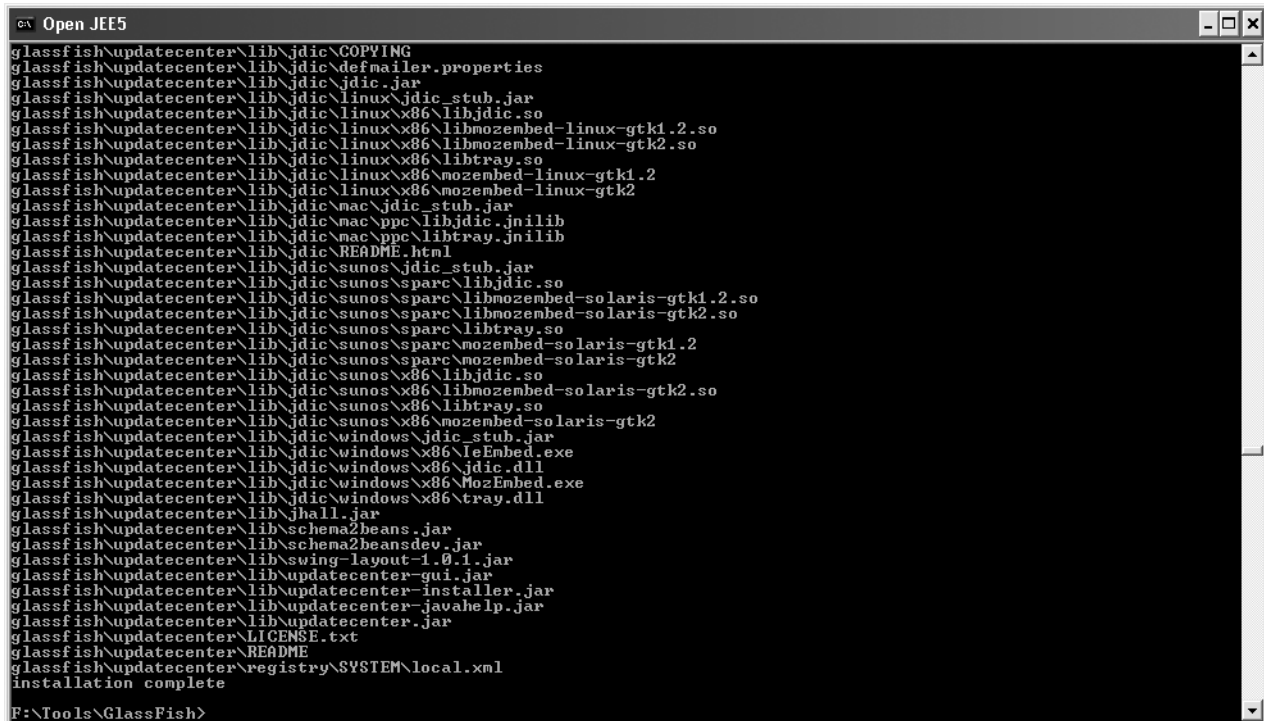


Figure 3-6 La décompression du fichier se fait avec succès.

La décompression s'effectue automatiquement dans le répertoire courant, jusqu'à l'apparition du message : *installation complete* (figure 3-6).

Il faut maintenant configurer GlassFish (assurez-vous d'avoir correctement positionné les variables `JAVA_HOME` et `ANT_HOME`). Pour ce faire, on utilise le fichier `Ant setup.xml` livré avec GlassFish. Tapez la commande suivante :

```
ant -f setup.xml
```

```

Open JCL5
do.chmod:
create-local-registry:
setup.reg:
[echo] Creating the service registry file..
[java] From Local service registry ..

get.java.home:
setup.init:
set.env.win:
set.env.unix:
set.env:
create.domain:
[exec] Using port 4848 for Admin.
[exec] Using port 8080 for HTTP Instance.
[exec] Using port 7676 for JMS.
[exec] Using port 3700 for IIOP.
[exec] Using port 8181 for HTTP_SSL.
[exec] Using default port 3820 for IIOP_SSL.
[exec] Using default port 3920 for IIOP_MUTUALAUTH.
[exec] Using default port 8686 for JMX_ADMIN.
[exec] Domain being created with profile:developer, as specified by variable AS_ADMIN_PROFILE in configuration file
.
[exec] The file in given locale [fr_FR] at: [F:\Tools\GlassFish\glassfish-v2ur2-b04\lib\install\templates\locales\fr_FR\index.html] could not be found. Using default [en_US] index.html instead.
[exec] Security Store uses: JKS
[exec] Domain domain1 created.
[exec] Admin login information for host [localhost] and port [4848] is being overwritten with credentials provided.
This is because the --savelogin option was used during create-domain command.
[exec] Login information relevant to admin user name [admin] for this domain [domain1] stored at [C:\Documents and Settings\Antonio\asadminpass1] successfully.
[exec] Make sure that this file remains protected. Information stored in this file will be used by asadmin commands to manage this domain.
[delete] Deleting: F:\Tools\GlassFish\glassfish-v2ur2-b04\passfile
BUILD SUCCESSFUL
Total time: 41 seconds
F:\Tools\GlassFish\glassfish-v2ur2-b04>

```

Figure 3-7 Configuration de GlassFish

La configuration s'effectue avec succès lorsque le message *build successful* s'affiche sur l'écran. Il faut alors positionner la variable `GLASSFISH_HOME` (par exemple, set `GLASSFISH_HOME=F:\Tools\Glassfish\glassfish-v2ur2-b04`) et rajouter le répertoire `bin` dans la variable système `PATH` (pour notre exemple : `PATH=%GLASSFISH_HOME%\bin;%PATH%`) (figure 3-7).

Pour vérifier que l'installation s'est bien déroulée, démarrez le domaine par défaut (`domain1`) de GlassFish en tapant la commande suivante :

```
asadmin start-domain domain1
```

Avec votre navigateur, rendez-vous à l'adresse `http://localhost:8080`. Si vous voyez la page d'accueil par défaut s'afficher, c'est que votre serveur fonctionne (figure 3-8).

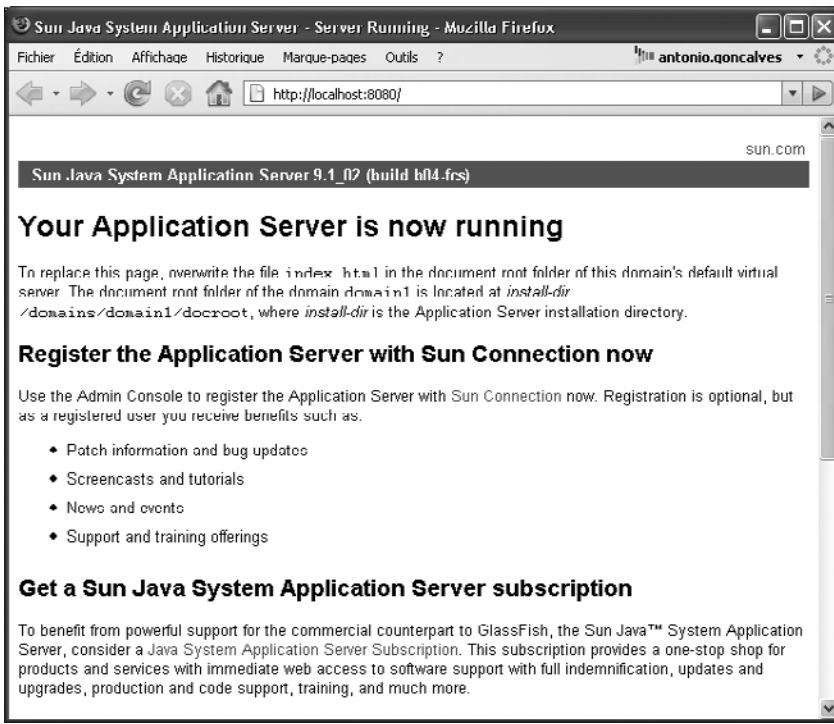


Figure 3–8
Page d'accueil par défaut de GlassFish

Pour arrêter le serveur par défaut `domain1`, tapez la commande :

```
asadmin stop-domain domain1
```

En parcourant l'arborescence des répertoires d'installation de GlassFish, on s'aperçoit que la base de données Derby se trouve dans `%GLASSFISH_HOME%\javadb`.

Configuration du serveur GlassFish

Nous venons d'installer les outils indispensables pour développer et exécuter l'application YAPS Pet Store. Il nous faut encore configurer GlassFish pour répondre aux besoins spécifiques de l'application.

L'utilitaire `asadmin`

La configuration de GlassFish peut se faire soit par une interface graphique, soit en ligne de commande via l'utilitaire `asadmin`. Celui-ci est fourni avec GlassFish (dans `%GLASSFISH_HOME%\bin`) et vous permet d'administrer les composants du plus simple (file d'attente JMS, sources de données) au plus complexe (clustering, haute disponibilité).

Outils Versions utilisées

Pour le développement de l'application YAPS Pet Store, voici la version des outils utilisés dans ce livre :

- JDK 1.5_15
- Ant 1.7
- GlassFish V2ur2-b04

GLASSFISH Administration avec asadmin

Pour faciliter la configuration du serveur GlassFish, les commandes `asadmin` seront incorporées dans des tâches Ant. Ainsi, vous n'aurez pas à saisir les commandes mais plutôt à utiliser les tâches se trouvant dans le fichier `admin.xml`. Celui-ci est téléchargeable sur le site
 ▶ <http://www.antoniogoncalves.org>

`asadmin` peut s'utiliser de deux manières :

- **en mode ligne de commande** : on invoque `asadmin` en lui passant des paramètres (par exemple, `asadmin ping-connection-pool --user admin --host localhost --port 8282 petstorePool`);
- **en mode shell** : en exécutant `asadmin` sans paramètre, on se retrouve dans un système de shell (reconnaissable à l'invite `asadmin>`) où l'on peut alors exécuter tous types de commandes et les enchaîner (par exemple, `ping-connection-pool --user admin --host localhost --port 8282 petstorePool`).

`asadmin` est un utilitaire extrêmement riche, mais on peut facilement obtenir de l'aide générale en utilisant la commande `asadmin help`, ou particulière à une commande (par exemple, `asadmin help ping-connection-pool`). Cerise sur le gâteau, `asadmin` comporte un système de suggestion de commande. Ainsi, si vous recherchez une commande qui agit sur la base de données (*database*) mais que vous ne vous souvenez plus de son nom, vous pouvez simplement taper `asadmin database`. L'utilitaire vous proposera alors les commandes `start-database` et `stop-database` (voir figure 3-9).

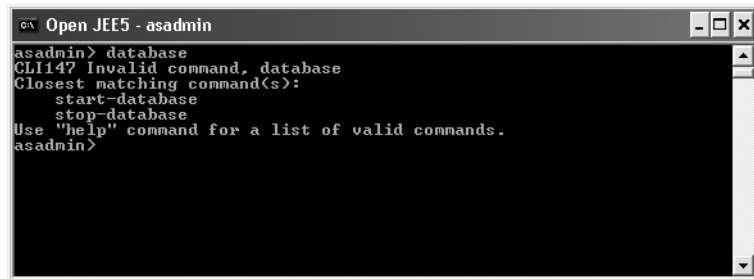


Figure 3-9

Aide en ligne de la commande `asadmin`

GLASSFISH Les mots de passe

Pour administrer GlassFish il faut, soit saisir le mot de passe administrateur à chaque commande en ligne, soit le stocker dans un fichier. Cette deuxième solution a été adoptée dans ce livre. Vous trouverez donc dans le code de YAPS Pet Store le fichier `passwordfile` qui contient le mot de passe administrateur sous le format :
`AS_ADMIN_PASSWORD=adminpwd`

Création d'un domaine

Tout d'abord, nous allons créer un domaine propre à l'application. Comme nous venons de le voir lors de l'installation, GlassFish possède un domaine par défaut qui se nomme `domain1`. Nous allons créer un domaine spécifique à YAPS Pet Store que nous nommerons `petstore`. Pour cela, tapez la commande suivante :

```
asadmin create-domain --adminport 8282 --user admin --savemasterpassword=true --instanceport 8080 petstore
```

L'utilitaire `asadmin` vous demande alors de saisir un mot de passe pour l'administrateur et un mot de passe pour le master. Pour simplifier l'administration de GlassFish, utiliser `adminpwd` pour l'administrateur et `masterpwd` pour le master.

```

F:\>asadmin create-domain --adminport 8282 --user admin --savemasterpassword=true --instanceport 8080 petstore
Please enter the admin password>
Please enter the admin password again>
Please enter the master password (Enter to accept the default!)>
Please enter the master password again (Enter to accept the default!)>
Using port 8282 for Admin.
Using port 8080 for HTTP Instance.
Using default port 2676 for JMS.
Using default port 3700 for IIOP.
Using default port 8181 for HTTP SSL.
Using default port 3820 for IIOP SSL.
Using default port 3920 for IIOP_MUTUALAUTH.
Using default port 8686 for JMX_ADMIN.
Domain being created with profile=developer, as specified by variable AS_ADMIN_PROFILE in configuration file.
The file in given locale (fr_FR) at: F:\Tools\GlassFish\glassfish-v2ur2-b04\lib\install\templates\locales\fr_FR\index.h
tmll could not be found. Using default (en_US) index.html instead.
Security Store uses: JMS
Domain petstore created.
F:\>_

```

Figure 3-10
Configuration
du domaine Pet Store

Cela a pour effet de créer un domaine intitulé `petstore` qui écoute sur le port 8080. Le port d'administration est le 8282. Vous trouverez donc le nouveau sous répertoire `%GLASSFISH_HOME%\domains\petstore`.

La commande `asadmin create-domain` que nous venons d'utiliser est la seule qui ne soit pas encapsulée dans une tâche Ant. Pour les commandes qui suivent, vous devez télécharger le code de l'application et utiliser le fichier `admin.xml`.

Démarrage du domaine

Pour configurer un domaine, le serveur doit être démarré. Pour cela, démarrez l'instance `petstore` à l'aide de la commande :

```
%PETSTORE_HOME%\> ant -f admin.xml start-domain
```

Assurez-vous que l'instance fonctionne en vous rendant à l'adresse `http://localhost:8080`. Pour accéder à la console d'administration, allez à l'adresse `http://localhost:8282` puis saisissez le nom de l'utilisateur `admin` et son mot de passe `adminpwd`. Pour arrêter l'instance, tapez la commande suivante :

```
%PETSTORE_HOME%\> ant -f admin.xml stop-domain
```

GLASSFISH Les ports d'écoute

GlassFish utilise plusieurs ports d'écoute par défaut pour traiter des requêtes ou des protocoles :

- 3700 : IIOP ;
- 3829 : IIOP sécurisé ;
- 4848 : console d'administration ;
- 8080 : HTTP ;
- 8181 : HTTPS ;
- 8686 : JMX.

Il est possible de créer plusieurs domaines sur un même serveur. Si tel est le cas, pour exécuter ces domaines en parallèle, les ports d'écoute par défaut doivent être

modifiés (un port ne peut être utilisé que par un seul process). Au lieu de les changer un par un pour chaque domaine, la commande `asadmin create-domain` permet de les modifier tous à la fois à partir d'un nombre de base. Il suffit d'utiliser le paramètre `-portbase` et le domaine est alors créé avec les règles suivantes :

- console d'administration = `portbase + 48` ;
- port HTTP = `portbase + 80` ;
- port IIOP = `portbase + 37` ;
- port JMX = `portbase + 86`.

```
asadmin create-domain -portbase 1000 --user admin
--savemasterpassword=true petstore
```

GLASSFISH Comprendre les domaines

GlassFish, comme bien d'autres serveurs d'applications, utilise le concept de domaines. Un domaine est une instance de serveur contenant ses propres fichiers de configuration. On peut ensuite y déployer plusieurs applications.

TÉLÉCHARGER YAPS Pet Store

Téléchargez le code de l'application, décompressez le fichier dans un répertoire et positionnez la variable `PETSTORE_HOME`.

► <http://www.antoniogoncalves.org>

ANT Tâche setup

Toutes les tâches de configuration que nous allons voir, sont regroupées dans la tâche `setup`. Ainsi, au lieu de les taper toutes une à une, vous pouvez utiliser :

```
ant -f admin.xml setup
```

Pool de connexions

Un pool de connexions est un mécanisme permettant de réutiliser les connexions à la base de données. Comme la création d'une nouvelle connexion JDBC consomme beaucoup de ressources, il est plus judicieux d'utiliser un pool qui réutilise les connexions libres.

Configuration de la base de données

Après avoir créé un domaine spécifique à l'application YAPS Pet Store, nous allons en faire de même pour la base de données Derby. Pour cela, il nous faut tout d'abord la démarrer à l'aide de la commande :

```
%PETSTORE_HOME%\> ant -f admin.xml start-db
```

Vous verrez alors apparaître un écran affichant les paramètres de Derby. Pour arrêter la base de données, tapez la commande suivante :

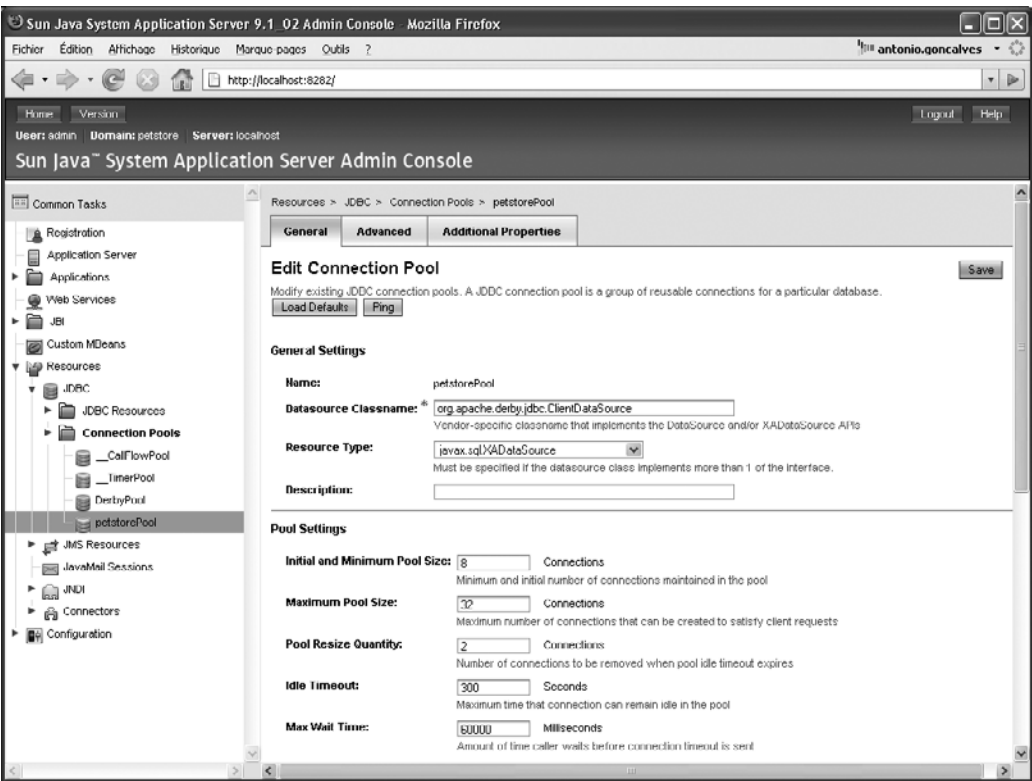
```
%PETSTORE_HOME%\> ant -f admin.xml stop-db
```

Création d'un pool de connexions

Dans un premier temps, nous allons créer un pool de connexions dans le serveur GlassFish (pour les tâches qui vont suivre, GlassFish et la base de données doivent être démarrés). Pour cela, utilisez la commande :

```
%PETSTORE_HOME%\> ant -f admin.xml create-connection-pool
```

Figure 3-11
Affichage du pool
de connexions dans la
console d'administration



Cela a pour effet de créer le pool de connexions `petstorePool` avec l'utilisateur `dbuser` et le mot de passe `dbpwd`. Pour vérifier que le pool est bien créé, vous pouvez soit utiliser la commande :

```
%PETSTORE_HOME%\> ant -f admin.xml list-connection-pool
```

soit utiliser la console d'administration (<http://localhost:8282>). Pour cela, naviguez dans l'arborescence du menu de droite et déployez les nœuds *Resources*, *JDBC*, *Connection Pools*. Vous verrez alors apparaître le pool `petstorePool` (figure 3–11).

Création de la base de données

Pour créer la base de données, nous utiliserons la flexibilité de Derby. En effet, il suffit de « pinger », c'est-à-dire lancer une commande `ping` sur le pool de connexions pour créer la base `petstoreDB`. Cette commande `ping` peut être faite soit via la console d'administration (cliquez sur le bouton *Ping* de l'écran précédent), soit par la commande suivante :

```
%PETSTORE_HOME%\> ant -f admin.xml ping-connection-pool
```

Vous verrez alors apparaître le nouveau répertoire `%GLASSFISH_HOME%\javadb\petstoreDB` où Derby stockera les données de l'application YAPS Pet Store. Pour accéder à cette base, utilisez le même utilisateur et mot de passe que celui défini dans le pool de connexions précédent (`dbuser/dbpwd`).

Création d'une source de données

Après le pool de connexions, il nous faut créer une source de données (*Data Source*). Comme nous le verrons au chapitre suivant *Objets persistants*, c'est cette source de données qui est référencée dans le code de l'application. Pour la créer, utilisez la commande :

```
%PETSTORE_HOME%\> ant -f admin.xml create-datasource
```

Pour vérifier que la source de données a bien été créée, vous pouvez soit utiliser la commande :

```
%PETSTORE_HOME%\> ant -f admin.xml list-datasource
```

soit utiliser la console d'administration (<http://localhost:8282>). Pour cela, naviguez dans l'arborescence du menu de droite et déployez les nœuds *Resources*, *JDBC*, *JDBC Resources*. Vous verrez alors la *DataSource jdbc/petstoreDS* (figure 3–12).

Une source de données

Une source de données (*Data Source*) propose de fournir une meilleure alternative à la classe `DriverManager` pour faciliter l'obtention d'une connexion à une base de données. Agissant comme une fabrique (*factory*), elle permet au serveur de contrôler le cycle de vie d'une connexion. L'utilisation d'un objet `DataSource` est obligatoire pour pouvoir utiliser un pool de connexions dans un serveur Java EE.

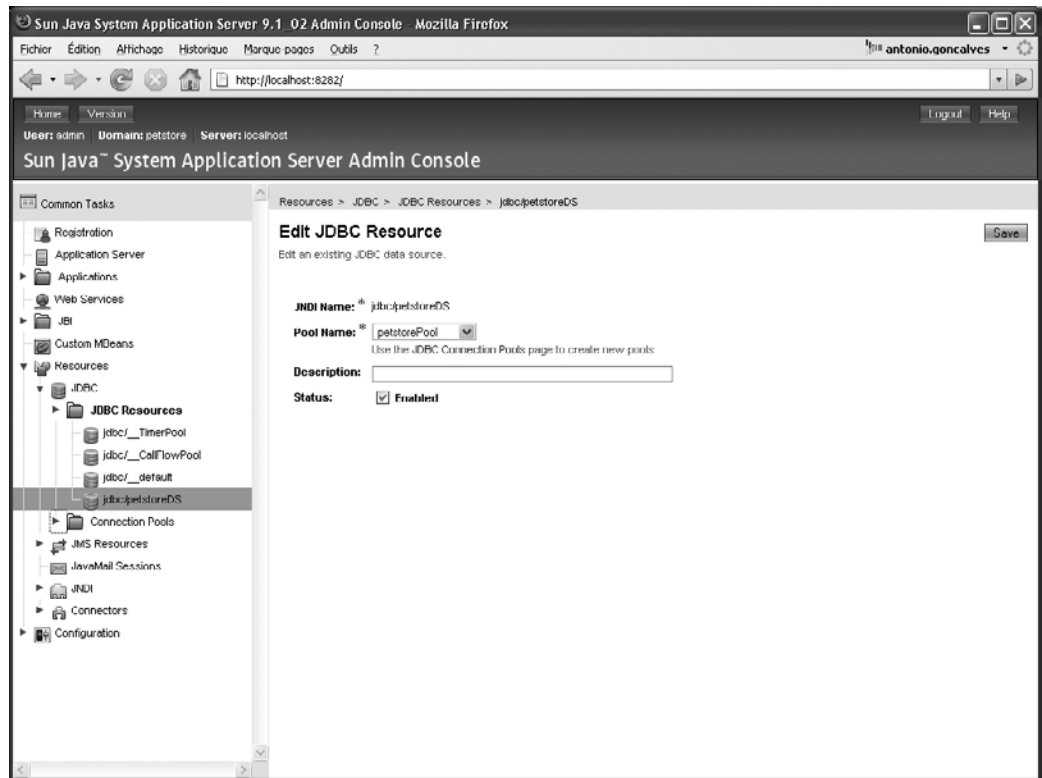


Figure 3–12
Affichage de la source de
données dans la console
d’administration

Création des ressources JMS

L’application YAPS Pet Store utilise JMS pour les traitements asynchrones. Il faut donc créer une fabrique de connexions ainsi qu’un Topic (file d’attente) JMS. Pour cela, tapez les commandes suivantes :

```
%PETSTORE_HOME%\> ant -f admin.xml create-jms-connection-factory
%PETSTORE_HOME%\> ant -f admin.xml create-jms-topic
```

Pour vérifier que toutes ces ressources ont bien été créées, tapez les commandes suivantes :

```
%PETSTORE_HOME%\> ant -f admin.xml list-jms-resources
```

Vous pouvez aussi utiliser la console d’administration (<http://localhost:8282>). Pour ce faire, naviguez dans l’arborescence du menu de droite et déployez les nœuds *Resources*, *JMS Resources*. Vous verrez les deux sous-menus *Connection Factories* (contenant la fabrique `jms/petstoreConnectionFactory`) et *Destination Resources* (contenant la file d’attente `jms/topic/order`), voir figure 3–13.

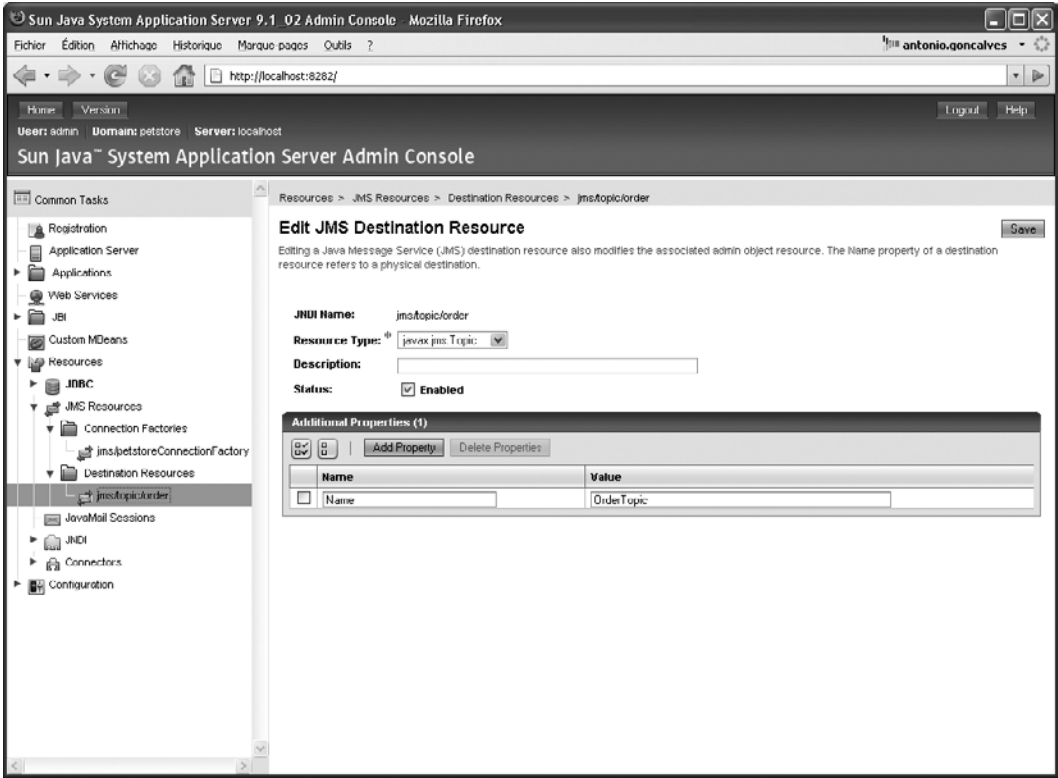


Figure 3–13
Affichage des ressources
JMS dans la console
d’administration

Création de loggers

Pour nous aider à déboguer l’application ou à suivre l’enchaînement de méthodes, nous utiliserons l’API de logging de Java. Pour cela, il nous faut créer différents loggers pour pouvoir consulter les traces dans la console de GlassFish :

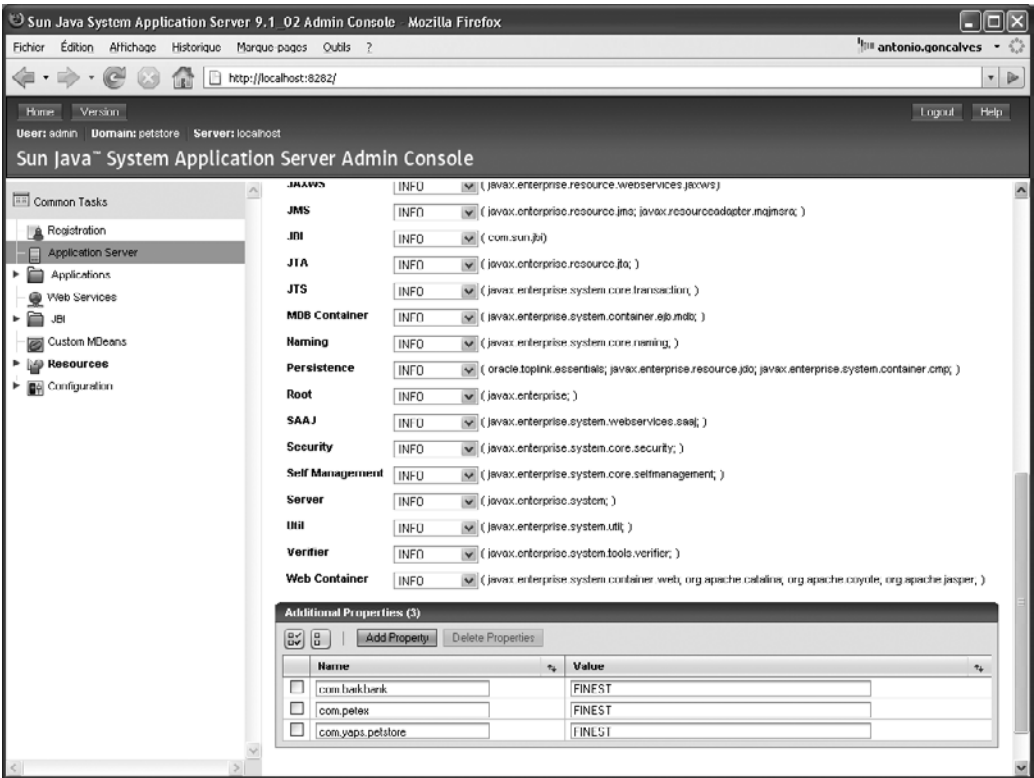
Tableau 3–1 Loggers

Nom du logger	Remarques
com.yaps.petstore	Traces de l’application YAPS Pet Store
com.barkbank.validator	Traces du service web de validation de carte bancaire de BarkBank
com.petex.transport	Traces du service web du transporteur PetEx

Ces loggers nous permettent d’ajouter des traces dans notre code et de les visualiser dans le fichier de log qui se trouve dans le répertoire %GLASSFISH_HOME%\domains\petstore\logs. On les crée à l’aide de la commande suivante :

```
%PETSTORE_HOME%\> ant -f admin.xml set-loggers
```

Figure 3–14
Affichage des loggers
en criticité Finest



RETOUR D'EXPÉRIENCE Les traces applicatives

Depuis le JDK 1.4, il existe une API de logging (paquetage `java.util.logging`). Cette API consiste à pister les traces des événements survenus dans un système ou dans une application. Un ou plusieurs fichiers de log au format prédéfini sont générés en cours d'exécution et conservent des messages informant sur la date et l'heure de l'événement, la nature de ce dernier ainsi que sa gravité par un code ou une description sémantique et éventuellement d'autres informations (utilisateur, classe, etc).

Pour logger un message, il faut utiliser les méthodes de l'objet `java.util.logging.Logger`. L'argument `Level` définit le niveau de criticité du message passé en paramètre. Si ce niveau est géré, le message sera redirigé vers tous les flux de sortie associés au journal. Il existe plusieurs niveaux :

- SEVERE : niveau le plus élevé.

- WARNING : avertissement.
- INFO : information.
- CONFIG : configuration.
- FINE : niveau faible.
- FINER : niveau encore plus faible.
- FINEST : niveau le plus faible.

Par exemple, pour logger le message « donnée invalide » avec un niveau de criticité avertissement, on utilise le code :

```
logger.log(Level.WARNING, "donnée invalide");
```

Cette API permet aussi d'éviter l'utilisation de la méthode `printStackTrace` d'une exception. Bien que très utile, cette méthode affiche la trace d'une exception à l'écran, sans la garder dans un fichier. En utilisant la méthode `throwing` de l'API de logging, la trace de l'exception pourra être répertoriée comme un message, c'est-à-dire dans un fichier ou sur la console.

Pour vérifier que les loggers ont bien été créés, utilisez la console d'administration (<http://localhost:8282>). Sélectionnez le menu *Application Server* puis cliquez sur les onglets *Logging* et *Log Levels*. En bas de la page, vous trouverez la section *Additional Module Log Level Properties* où vous verrez apparaître les noms des loggers (figure 3–14).

Récapitulatif des éléments de configuration

Nous venons de voir comment configurer le serveur GlassFish et les différents composants qui seront utilisés par l'application. Le tableau ci-après nous donne un récapitulatif de cette configuration.

Tableau 3–2 Configuration GlassFish

Élément	Description
petstore	Nom du domaine GlassFish utilisé par l'application
http://localhost:8080	URL de l'application
http://localhost:8282	URL de la console d'administration
admin/adminpwd	Login et mot de passe pour accéder à la console d'administration GlassFish
master/masterpwd	Login et mot de passe du super utilisateur GlassFish
petstoreDB	Nom de la base Derby où seront stockées les données de l'application
dbuser/dbpwd	Login et mot de passe de la base de données
petstorePool	Nom du pool de connexions à la base
petstoreDS	Nom de la source de données pour accéder à la base
jms/petstoreConnectionFactory	Fabrique JMS
jms/topic/order	File d'attente JMS pour la gestion des bons de commande
com.yaps.petstore	Nom du logger de l'application YAPS Pet Store
com.barkbank.validator	Logger du service web de validation de carte bancaire de BarkBank
com.petex.transport	Logger du service web du transporteur PetEx

Environnement de développement

Tout au long de ce livre nous développerons trois applications :

- **YAPS Pet Store** : le site de commerce électronique permettant d'acheter des animaux de compagnie ;
- **BarkBank** : composant de validation des cartes bancaires ;
- **PetEx** : application permettant au transporteur de livrer les animaux domestiques aux différents clients.

Ces applications auront des arborescences différentes et seront déployées dans des fichiers .ear (ou .war) séparés.

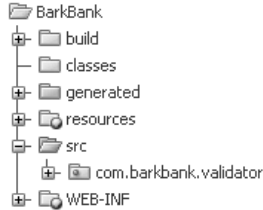
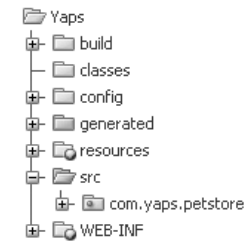
Les fichiers .ear

Le fichier .ear (Enterprise Archive) est une archive au format JAR qui contient tous les fichiers d'une application Java EE (classes Java, EJB, pages web, images...). Les fichiers .ear sont déployés et exécutés par les serveurs d'applications.

Les répertoires

Vous pourrez utiliser l'IDE de votre choix pour développer ces applications. Cependant, afin d'utiliser les tâches Ant, il faudra respecter l'arborescence des différents répertoires. La racine se trouve dans le répertoire %PETSTORE_HOME%.

Tableau 3-3 Répertoires des applications

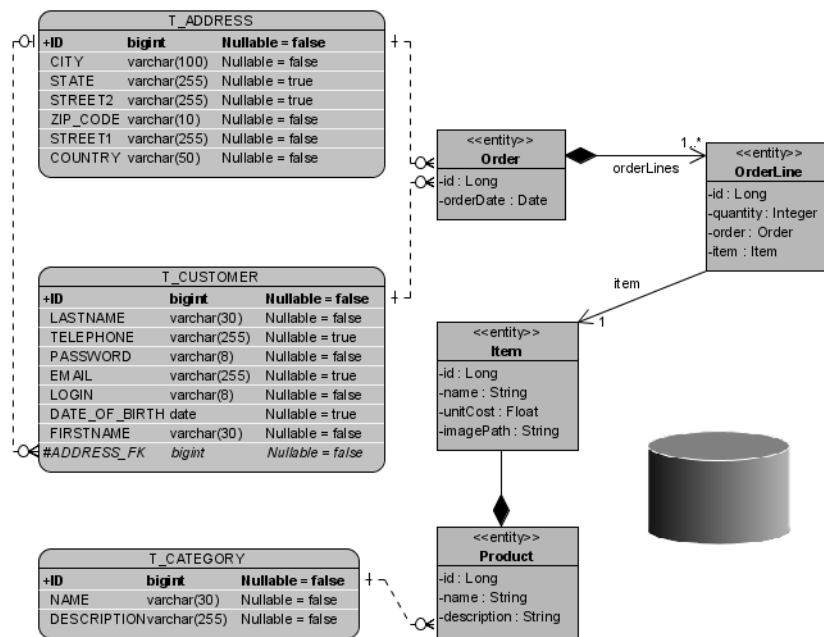
Élément	Description
	Le code de l'application BarkBank se trouve dans le répertoire %PETSTORE_HOME%\BarkBank. Vous y trouverez les sous-répertoires suivants : build : contient le fichier .war à déployer dans GlassFish ; classes : répertoire contenant le bytecode des classes de l'application ; generated : classes Java générées pour les services web ; resources : pages web et images du site de BarkBank ; src : ce répertoire contient les fichiers source Java de l'application de validation des cartes bancaires. Les classes sont sous le paquetage com.barkbank.validator ; WEB-INF : les fichiers de configuration de l'application web.
	Le code de l'application PetEx se trouve dans le répertoire %PETSTORE_HOME%\PetEx. Vous y trouverez les sous-répertoires suivants : build : contient le fichier .war à déployer dans GlassFish ; classes : répertoire contenant le bytecode des classes de l'application ; generated : classes Java générées pour les services web ; resources : pages web et images du site de PetEx ; src : ce répertoire contient les fichiers source Java de l'application du transporteur. Les classes sont sous le paquetage com.petex.transport ; WEB-INF : les fichiers de configuration de l'application web.
	Le code de l'application YAPS Pet Store se trouve dans le répertoire %PETSTORE_HOME%\Yaps. Vous y trouverez les sous-répertoires suivants : build : contient le fichier .ear à déployer dans GlassFish ; classes : répertoire contenant le bytecode des classes de l'application ; config : contient les fichiers de paramétrage de l'application ; generated : classes Java générées pour les services web ; resources : pages web et images du site YAPS Pet Store ; src : ce répertoire contient les fichiers source Java de l'application YAPS Pet Store. Les classes sont sous le paquetage com.yaps.petstore ; WEB-INF : les fichiers de configuration de l'application web.
	À la racine, vous trouverez les deux fichiers des tâches Ant : admin.xml : toutes les tâches d'administration du serveur GlassFish sont regroupées dans ce fichier. build.xml : tâches pour compiler et déployer les applications.

En résumé

Ce chapitre nous a présenté les outils utilisés pour le développement de l'application YAPS Pet Store, c'est-à-dire Ant, le JDK, le serveur d'applications GlassFish et la base de données Derby. Nous les avons installés puis configurés pour répondre à nos besoins techniques. Les développements peuvent donc commencer.

4

chapitre



Objets persistants

Nous commencerons les développements par la couche de persistance. Après avoir présenté brièvement ce concept, ce chapitre se concentre sur JPA (Java Persistence API). Il décrit les annotations élémentaires du mapping objet-relationnel et les annotations avancées (relations, jointures, cascades, etc.). À partir de l'étude de cas abordée dans le premier chapitre, nous définirons et implémenterons les entités de l'application YAPS Pet Store.

SOMMAIRE

- ▶ Couche de persistance
- ▶ JPA et les entités
- ▶ Cycle de vie des entités
- ▶ Mapping objet-relationnel
- ▶ Les relations 0:1, 1:1 et 1:n
- ▶ Les objets persistants de l'application
- ▶ Schéma de la base de données

MOTS-CLÉS

- ▶ JDBC
- ▶ JPA
- ▶ Entity
- ▶ Annotations
- ▶ Relations
- ▶ Mapping O-R
- ▶ DDL

/// JVM

La machine virtuelle Java est une surcouche logicielle spécifique à chaque système d'exploitation pour interpréter le code Java.

/// SQL

Structured Query Language (SQL), traduisez langage structuré de requêtes, est un langage destiné à interroger une base de données.

/// Le design pattern DAO

Le design pattern DAO, (*Data Access Object*), est fréquemment utilisé pour simplifier la programmation avec JDBC. Il permet de déléguer la persistance d'un objet métier vers un objet DAO. Ce dernier peut utiliser une base de données, un fichier texte, une base objet ou même un serveur LDAP.

Composé de trois objets, l'interface, la fabrique (ou *Factory*) et l'implémentation, le design pattern DAO propose des méthodes pour manipuler les données (récupérations et modifications).

La persistance des données

Le langage Java instancie des objets en mémoire et les manipule au travers de méthodes modifiant ainsi leur état. Cet état n'est cependant accessible que lorsque la JVM (*Java Virtual Machine*) s'exécute : si celle-ci s'arrête, le contenu de la mémoire disparaît ainsi que les objets et leur état. L'un des fondements de la programmation consiste à réutiliser ces données. On appelle cela la persistance des données.

La persistance est ainsi le fait d'exister dans le temps. Un objet qui reste en l'état lorsqu'il est sauvegardé, puis rechargé plus tard, possède la propriété de persistance. Le langage Java, d'une part, et certains frameworks, d'autre part, nous permettent de rendre persistants les objets de différentes manières.

La sérialisation

Au travers du mécanisme de sérialisation, Java fournit une méthode simple, transparente et standard pour réaliser la persistance. Le format utilisé étant indépendant du système d'exploitation, un objet sérialisé sur un système peut être réutilisé par un autre système.

Les objets peuvent être sérialisés sur le disque dur ou sur le réseau (Internet compris). Pour qu'un objet soit sérialisable, il doit implémenter l'interface `java.io.Serializable` et posséder des attributs sérialisables : Java saura alors comment rendre l'objet persistant.

Ce mécanisme, bien que très simple et directement utilisable par le langage Java, est rarement employé pour des applications d'une certaine envergure. Il n'offre ni langage de requête, ni d'infrastructure professionnelle permettant la résistance aux fortes charges.

JDBC

JDBC (*Java Data Base Connectivity*) est la couche logicielle standard offerte aux développeurs pour accéder à des bases de données relationnelles. Elle se charge de trois étapes indispensables à l'accès aux données :

- la création d'une connexion à la base ;
- l'envoi d'instructions SQL ;
- l'exploitation des résultats provenant de la base.

Cette API fait partie intégrante de la plate-forme Java depuis la version 1.1 du JDK. Elle est représentée par le paquetage `java.sql`. Bien qu'encore largement utilisée, elle a tendance à disparaître au profit d'outils de mapping objet-relationnel. En effet, développer une couche d'accès aux

données avec JDBC, même utilisée de pair avec le pattern DAO, est un travail fastidieux, répétitif, qui consomme beaucoup de temps.

Mapping objet-relationnel

Le principe du mapping objet-relationnel (ORM ou *object-relational mapping*) consiste à déléguer l'accès aux données à des outils ou frameworks externes. Son avantage est de proposer une vue orientée objet d'une structure de données relationnelle (lignes et colonnes).

Les outils de mapping mettent en correspondance bidirectionnelle les données de la base et les objets. Pour cela, ils utilisent des mécanismes de configuration pour exécuter les requêtes SQL.

Il existe plusieurs API et frameworks permettant de faire du mapping objet-relationnel : les entity beans 2.x, Hibernate, TopLink, JDO et JPA. Pour stocker les données de l'application YAPS Pet Store, le choix se porte logiquement sur JPA, la nouvelle API de persistance de Java Enterprise Edition.

Java Persistence API

La persistance des données en Java EE 5 a été complètement réarchitecturée au travers de JPA (Java Persistence API). Alors que nous parlions de composants persistants en EJB 2.x (entity beans), JPA se recentre sur de simples classes Java (entities). En EJB 2.x, la persistance ne pouvait être assurée qu'à l'intérieur du conteneur alors qu'avec JPA, elle peut être utilisée dans une simple application Java SE (*Java Standard Edition*). Il fallait auparavant utiliser le mécanisme complet de création des EJB pour obtenir un entity bean (au travers des interfaces Home, Local ou Remote) alors que maintenant on utilise tout simplement l'opérateur `new`. Dans la version 2.x, les possibilités du mapping O-R étaient limitées alors qu'avec JPA on peut maintenant mettre en œuvre les notions d'héritage et de multitable (les attributs d'un objet peuvent être stockés dans plus d'une table).

JPA est une abstraction au-dessus de JDBC et permet de s'affranchir du langage SQL. Toutes les classes et annotations de cette API se trouvent dans le paquetage `javax.persistence`.

PERSISTANCE **Hibernate**

Hibernate est un framework Open Source destiné à gérer la couche d'accès aux données. Pour ceux d'entre vous qui connaissent ce framework, vous remarquerez que JPA s'en est très fortement inspiré. La grande nouveauté étant le système d'annotations qui permet de se passer de fichiers de configuration XML (fichier `.hbm` dans hibernate). On peut retrouver ce mécanisme en mêlant Hibernate et xDoclet.

Depuis sa version 3.2, Hibernate est compatible avec JPA.

- ▶ <http://www.hibernate.org/>
- ▶ <http://xdoclet.sourceforge.net/>

EJB **Les entity beans 2.x**

Pour vous faire une idée des modifications apportées à la spécification EJB, retrouvez en annexe le code source d'un entity bean 2.1.

PERSISTANCE **Implémentations JPA**

Cet ouvrage utilise TopLink Essentials comme implémentation de JPA, mais il en existe d'autres comme Hibernate, Kodo ou OpenJPA. Chacune de ces implémentations se doit de suivre la spécification, mais peut apporter quelques atouts spécifiques, comme la gestion du cache, qui améliore les performances.

TopLink Essentials est l'implémentation de référence de JPA 1.0.

- ▶ <http://openjpa.apache.org/>
- ▶ <http://www.hibernate.org/>
- ▶ <http://www.oracle.com/technology/products/ias/toplink/jpa/index.html>
- ▶ <http://www.bea.com/kodo>

Entity Bean et Entity

Les EJB 2.x utilisent le terme d'*entity bean* pour définir les composants persistants. Avec JPA, on parle plutôt tout simplement d'*entity* (ou entité).

Rappel Pojo

Pojo est l'acronyme de *Plain Old Java Object*, que l'on pourrait traduire par « bon vieil objet Java ».

SYNTAXE Entity

Un entity est une classe Java qui possède les caractéristiques suivantes :

- elle doit être annotée par `@Entity` ;
- elle doit avoir un identifiant annoté par `@Id` ;
- elle doit être publique ;
- elle ne doit pas être finale ;
- elle peut implémenter `Serializable` seulement si elle doit traverser des couches réseau ;
- elle ne doit pas définir la méthode `finalize` ;
- elle doit avoir un constructeur par défaut.

REMARQUE Constructeur par défaut

Une classe Java qui ne déclare pas explicitement de constructeur possède tout de même un constructeur par défaut.

Entity

Dans le modèle de persistance JPA, un entity est une simple classe Java (Pojo). On déclare, instancie et utilise cet entity tout comme n'importe quelle autre classe. Un entity possède des attributs (son état) qui peuvent être manipulés via des accesseurs (méthodes `get` et `set`). Grâce aux annotations, ces attributs peuvent être rendus persistants en base de données.

Exemple d'entity

Rien ne vaut un premier exemple simple d'entity pour expliquer le mapping objet-relationnel.

Exemple simple d'entity

```
@Entity ❶
public class Address {

    @Id ❷
    private Long id;
    private String street1;
    private String street2;
    private String city;
    private String state;
    private String zipcode;
    private String country;
    // Accesseurs get/set
}
```

Cet exemple de code représente une classe `Address`. Notez la présence d'annotations à plusieurs endroits dans la classe. Tout d'abord, l'annotation `@javax.persistence.Entity` ❶ permet à JPA de reconnaître cette classe comme une classe persistante et non comme une simple classe Java. L'annotation `@javax.persistence.Id` ❷, quant à elle, définit l'identifiant unique de l'objet. Elle donne à l'entity une identité en mémoire en tant qu'objet, et en base de données via une clé primaire. Les autres attributs (`street1`, `street2`,... `country`) seront rendus persistants par JPA en appliquant les paramètres par défaut : le nom de la colonne est identique à celui de l'attribut et le type `String` est converti en `varchar(255)`.

Cet exemple ne comporte que des attributs, mais la classe peut aussi avoir des méthodes métier comme nous le verrons par la suite. Notez que cet entity `Address` est une simple classe Java (Pojo). Elle n'implémente aucune interface, se contente d'être annotée par `@javax.persistence.Entity` et d'avoir un identifiant unique (`@javax.persistence.Id`). Pour être un entity, une classe doit au minimum utiliser ces deux annotations et posséder un constructeur par défaut.

Grâce à ces annotations, JPA peut synchroniser les données entre les attributs de l'entity `Address` et les colonnes de la table `Address`. Ainsi, si l'attribut `zipcode` est modifié par l'application, JPA se chargera de modifier cette valeur dans la colonne `zipcode`.

DDL de la table `ADDRESS`

```
CREATE TABLE ADDRESS (  
  
    ID BIGINT NOT NULL,  
    CITY VARCHAR(255),  
    STATE VARCHAR(255),  
    STREET2 VARCHAR(255),  
    ZIPCODE VARCHAR(255),  
    STREET1 VARCHAR(255),  
    COUNTRY VARCHAR(255),  
  
    PRIMARY KEY (ID)  
)
```

- Le nom de la table est identique à celui de la classe.
- Les attributs de la classe sont stockés dans des colonnes qui portent le même nom que les attributs de l'entity.
- L'attribut `id` est la clé primaire de la table.

DDL

Le langage de définition de données (*Data Definition Language* ou DDL) permet de manipuler les structures de données et non les données elles-mêmes. Dans notre exemple, l'ordre `Create Table` crée la structure d'une table dans la base.

REMARQUE Génération des DDL

Les DDL peuvent, soit être écrites par un DBA (administrateur de base de données), soit être générées automatiquement à partir des annotations JPA. Comme nous le verrons dans le chapitre 6, *Exécution de l'application*, le cas décrit dans cet ouvrage utilise la génération automatique.

Annotations élémentaires du mapping

L'exemple simple de l'entity `Address` peut être complété pour modifier certaines conventions de nommage (tables, colonnes) ou de typage (colonne non nulle, de longueur définie, etc.). Pour cela, il suffit d'utiliser les attributs des annotations JPA.

Table

L'annotation `@javax.persistence.Table` permet de définir les valeurs par défaut liées à la table. Elle n'est pas obligatoire mais permet, par exemple, de spécifier le nom de la table dans laquelle les données seront stockées. Si cette annotation est omise, le nom de la table sera le même que celui de la classe.

ANNOTATIONS Les descripteurs XML

Pour ceux qui sont habitués à utiliser des descripteurs XML en lieu et place des annotations, JPA vous laisse le choix. Cependant, cet ouvrage ne couvre pas les descripteurs XML et se concentrera sur les annotations.

Définir une annotation

Les annotations sont définies par des méta-annotations. Par exemple, le code de l'annotation `@Table` que nous venons de voir, utilise plusieurs méta-annotations.

`@Target` permet de limiter le type d'éléments sur lesquels l'annotation peut être utilisée. Si elle est absente de la déclaration, elle peut alors être utilisée sur tous ces éléments :

- `ANNOTATION_TYPE` : l'annotation peut être utilisée sur d'autres annotations.
- `CONSTRUCTOR` : l'annotation peut être utilisée sur des constructeurs.
- `FIELD` : l'annotation peut être utilisée sur des champs d'une classe.
- `LOCAL_VARIABLE` : l'annotation peut être utilisée sur des variables locales.
- `METHOD` : l'annotation peut être utilisée sur des méthodes.

Persister les données d'une classe dans plusieurs tables

Par défaut, JPA suppose que tous les champs persistants d'un entity sont stockés dans une seule table. Cette table est connue comme la table primaire (annotation `@Table`). Dans certaines situations, vous pouvez souhaiter persister les données d'une classe dans une table primaire ainsi que dans une ou plusieurs tables secondaires. Pour cela, il faut employer l'annotation `@SecondaryTable` pour associer un Entity à une table secondaire ou `@SecondaryTables` pour plusieurs tables secondaires. JPA persistera alors certains champs dans la table primaire et certains autres dans les tables secondaires. Dans ce cas, il faut associer les attributs à une des tables en utilisant l'annotation `@Column` (que l'on verra dans les prochaines pages). Ci-dessous une classe `Address` persistant ses données dans une table primaire et deux tables secondaires :

```
@Entity
@Table(name = "t_address") ❶
@SecondaryTables({ ❷
    @SecondaryTable(name="t_city"), ❸
    @SecondaryTable(name="t_country") ❹
})
public class Address {

    @Id
    private Long id;
    private String street1; ❺
    private String street2; ❺
    @Column(table="t_city")
    private String city; ❻
    @Column(table="t_city")
    private String state; ❻
    @Column(table="t_city")
```

- `PACKAGE` : l'annotation peut être utilisée sur des paquets.
- `PARAMETER` : l'annotation peut être utilisée sur des paramètres d'une méthode ou d'un constructeur.
- `TYPE` : l'annotation peut être utilisée sur la déclaration d'un type : classe, interface (annotation comprise) ou énumération.

`@Retention` indique la manière dont l'annotation doit être gérée par le compilateur. Elle peut prendre une de ces trois valeurs :

- `SOURCE` : les annotations sont présentes dans le source mais ne sont pas enregistrées dans le fichier `.class`.
- `CLASS` : les annotations sont enregistrées dans le fichier `.class` à la compilation mais elle ne sont pas utilisées par la machine virtuelle à l'exécution de l'application.
- `RUNTIME` : les annotations sont enregistrées dans le fichier `.class` à la compilation et sont utilisées par la machine virtuelle à l'exécution de l'application.

```
private String zipcode; ❻
@Column(table="t_country")
private String country; ❼
// Accesseurs get/set
}
```

Par défaut, les attributs ❺ de la classe `Address` sont persistés dans la table primaire `t_address` ❶. L'annotation `@SecondaryTables` ❷ signale à JPA qu'il existe deux tables secondaires : `t_city` ❸ et `t_country` ❹. Il suffit ensuite de spécifier à l'aide de l'annotation `@Column` les attributs qui sont stockés dans `t_city` ❻ ou `t_country` ❼.

Le résultat est donc la création de trois tables contenant des attributs différents et une même clé primaire.

```
CREATE TABLE T_ADDRESS (
    ID BIGINT NOT NULL,
    STREET1 VARCHAR(255),
    STREET2 VARCHAR(255),
    PRIMARY KEY (ID)
)
CREATE TABLE T_CITY (
    ID BIGINT NOT NULL,
    CITY VARCHAR(255),
    STATE VARCHAR(255),
    ZIPCODE VARCHAR(255),
    PRIMARY KEY (ID)
)
CREATE TABLE T_COUNTRY (
    ID BIGINT NOT NULL,
    COUNTRY VARCHAR(255),
    PRIMARY KEY (ID)
)
```

Code de l'annotation @javax.persistence.Table

```
package javax.persistence;

@Target({TYPE}) @Retention(RUNTIME)
public @interface Table {

    String name() default "";

    String catalog() default "";
    String schema() default "";

    UniqueConstraint[] uniqueConstraints() default {};

}
```

- ◀ Cette annotation s'applique à une classe.
- ◀ Nom de la table.
- ◀ Identifie le catalogue et le schéma de la base de données relationnelle.
- ◀ Ce tableau permet de définir les contraintes d'unicité sur une ou plusieurs colonnes.

Ainsi, si nous voulions changer le nom de la table en `t_address`, nous devrions écrire le code suivant :

Entity Address persistant ses données dans la table `t_address`

```
@Entity
@Table(name = "t_address")
public class Address {

    @Id
    private Long id;
    private String street1;
    // (...)

}
```

Clé primaire

Comme nous l'avons vu précédemment, un entity doit avoir au minimum les annotations `@Entity` et `@Id`. `@javax.persistence.Id` annote un attribut comme étant un identifiant unique.

Code de l'annotation @javax.persistence.Id

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)

public @interface Id {}
```

- ◀ Cette annotation s'applique à une méthode ou un attribut.
- ◀ L'annotation `@Id` n'a pas de méthode.

La valeur de cet identifiant peut être, soit générée manuellement par l'application, soit générée automatiquement grâce à l'annotation `@javax.persistence.GeneratedValue`. Trois valeurs sont alors possibles :

- La génération de la clé unique se fait de manière automatique (AUTO) par la base de données (valeur par défaut).
- On utilise une séquence SQL (SEQUENCE) pour obtenir cette valeur.
- Les identifiants sont stockés dans une table (TABLE).

Cette annotation s'applique à une méthode ou un attribut.

La génération de la clé unique peut être faite soit de manière automatique (AUTO), soit en utilisant une séquence SQL (SEQUENCE) ou une table contenant les identifiants (TABLE).

Code de l'annotation `@javax.persistence.GeneratedValue`

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)

public @interface GeneratedValue {

    GenerationType strategy() default AUTO;

    String generator() default "";

}
```

Ci-après le code de la classe `Address` avec une génération automatique de l'identifiant :

Entité `Address` avec génération automatique d'identifiant

```
@Entity
@Table(name = "t_address")
public class Address {

    @Id
    @GeneratedValue (strategy = GenerationType.AUTO)
    private Long id;
    private String street1;
    // (...)
}
```

ANNOTATIONS **Attribut ou méthode**

La plupart des annotations peuvent s'appliquer sur les attributs ou sur les méthodes. Par exemple, l'annotation `@Id` peut être accolée à l'attribut `id` ou à la méthode `getId()`. Dans cet ouvrage, par choix mais aussi pour faciliter la lecture, les annotations sont appliquées aux attributs et non aux méthodes.

Cette annotation s'applique à une méthode ou à un attribut.

Nom de la colonne.

La valeur doit-elle être unique ?

La valeur `null` est-elle autorisée ?

Autorise-t-on la colonne dans un ordre `insert` ou `update` ?

Colonne

L'annotation `@javax.persistence.Column` définit les propriétés d'une colonne. On peut ainsi changer son nom (qui par défaut porte le même nom que l'attribut), préciser son type, sa taille et si la colonne autorise ou non la valeur `null`.

Code de l'annotation `@javax.persistence.Column`

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)

public @interface Column {

    String name() default "";

    boolean unique() default false;

    boolean nullable() default true;

    boolean insertable() default true;
    boolean updatable() default true;

}
```

```
String columnDefinition() default "";

String table() default "";

int length() default 255;

int precision() default 0;
int scale() default 0;
}
```

- ◀ Cet attribut peut contenir la définition de la colonne au format DDL.
- ◀ Utilisé lors d'un mapping multitable.
- ◀ Longueur maximale pour une colonne de type Varchar.
- ◀ Pour les colonnes de type numérique, on peut rajouter la précision.

Ainsi, pour redéfinir les valeurs par défaut de l'entity Address, on peut utiliser l'annotation `@Column` de différentes manières :

Entity Address avec redéfinition des colonnes

```
@Entity
@Table(name = "t_address")
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String street1;
    private String street2;

    @Column(nullable = false, length = 100)
    private String city;
    private String state;

    @Column(name = "zip_code", nullable = false, length = 10)
    private String zipcode;

    @Column(nullable = false, length = 50)
    private String country;

    // accesseurs get/set
    public Long getId() {return id;}

    public String getStreet1() {return street1;}
    public void setStreet1(String street1) {
        this.street1 = street1;
    }

    public String getStreet2() {return street2;}
    public void setStreet2(String street2) {
        this.street2 = street2;
    }
}
```

- ◀ L'entity est stocké dans la table `t_address`.
- ◀ L'attribut `id` est l'identifiant de cet entity. Sa valeur est générée automatiquement par la base de données.
- ◀ L'attribut `street1` ne peut être `null`.
- ◀ L'attribut `city` ne peut être `null` et sa longueur maximale est de 100 caractères.
- ◀ L'attribut `zipcode` est mappé dans la colonne `zip_code` de 10 caractères de long.
- ◀ Il n'y a pas de méthode `setId` puisque ce n'est pas l'application qui génère l'identifiant mais la base de données.
- ◀ Accesseurs des attributs.

REMARQUE Un setter pour l'identifiant ?

Notez que, dans notre exemple, il n'y a pas de méthode `setId()`. En effet, l'identifiant est généré automatiquement grâce à l'annotation `@GeneratedValue`. Nous n'avons pas besoin de pouvoir changer cette valeur.

ALTERNATIVE Mapping par XML

Ce chapitre vous présente différentes annotations vous permettant de personnaliser un mapping objet-relationnel (changer le nom des tables, le nom des colonnes, leur type...). Il faut savoir que toutes ces opérations sont également réalisables via une configuration XML. En fait, il vous est même possible de panacher annotations et XML, sachant que le XML prendra le dessus sur les annotations.

Le mapping XML n'est pas abordé dans cet ouvrage.

Cette annotation s'applique à une méthode ou un attribut.

Trois types de dates possibles : `DATE`, `TIME` et `TIMESTAMP` (valeur par défaut).

DDL obtenu pour cet entity

```
CREATE TABLE T_ADDRESS (
  ID BIGINT NOT NULL,
  CITY VARCHAR(100) NOT NULL,
  STATE VARCHAR(255),
  STREET2 VARCHAR(255),
  ZIP_CODE VARCHAR(10) NOT NULL,
  STREET1 VARCHAR(255) NOT NULL,
  COUNTRY VARCHAR(50) NOT NULL,
  PRIMARY KEY (ID)
)
```

Comme nous pouvons le constater, JPA permet de modifier la DDL de la table au travers des annotations.

Annotations avancées

Date et heure

En Java, nous pouvons utiliser les classes `java.util.Date` ou `java.util.Calendar` pour représenter une date ou une heure. Lors du mapping objet-relationnel, on peut spécifier ce type grâce à l'annotation `@javax.persistence.Temporal`. Elle peut prendre trois valeurs possibles : `DATE`, `TIME` ou `TIMESTAMP`, qui est la valeur par défaut.

Code de l'annotation `@javax.persistence.Temporal`

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)

public @interface Temporal {

    TemporalType value();

}
```

RETOUR D'EXPÉRIENCE Générer la base de données

Avec JPA, il est possible d'utiliser trois approches pour gérer le mapping entre objets et base de données.

La première consiste à partir des entités pour générer le schéma de la base (ce que nous faisons dans cet ouvrage). Cette option n'est possible que lorsque le projet n'a pas de base de données existante et qu'il est de petite taille. En effet, dans la plupart des gros projets, il y a un DBA (*Database administrator* – Administrateur de base de données) qui contrôle que la structure de la base répond aux contraintes de performances exigées. Il faut alors adapter le mapping.

La deuxième consiste à générer les entités à partir d'une base de données existante. On se retrouve alors avec un modèle qui se cale très bien sur les données, mais qui n'a plus grand chose d'objet (pas d'abstraction, parfois même pas d'héritage).

La troisième approche, qui est la plus adoptée, consiste à ne rien générer mais à utiliser la puissance des annotations pour caler un modèle objet sur un modèle relationnel. Chaque monde a ses avantages et ses contraintes, ils doivent s'influencer le moins possible, et les outils de mapping sont justement là pour ça.

Ci-après un extrait de code de la classe client avec un attribut date de naissance de type `@Temporal (TemporalType.DATE)`.

Entity Customer avec date de naissance de type `@Temporal`

```
@Entity
@Table(name = "t_customer")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "date_of_birth")
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    // (...)
}
```

Remarquez que dans cet exemple, l'attribut `dateOfBirth` (date de naissance) est annoté deux fois. `@Column` permet de renommer la colonne en `date_of_birth` et `@Temporal` de préciser le format `DATE`. Comme nous le verrons par la suite, il est courant d'annoter un attribut à l'aide de plusieurs annotations.

Données non persistées

Avec JPA, dès qu'une classe est annotée persistante (`@Entity`), ses attributs sont tous automatiquement stockés dans une table. Si l'on veut qu'un attribut ne soit pas rendu persistant, on doit utiliser l'annotation `@javax.persistence.Transient`.

Par exemple, l'âge du client n'a pas besoin d'être rendu persistant en base puisqu'il peut être calculé à partir de la date de naissance.

Entiy Customer avec un attribut `Transient`

```
@Entity
@Table(name = "t_customer")
public class Customer {
    (...)

    @Column(name = "date_of_birth")
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;

    @Transient
    private Integer age;
}
```

REMARQUE **Mot-clé Java transient**

Avant l'apparition des annotations, Java introduisait déjà le mot-clé `transient` qui précise que l'attribut ne doit pas être inclus dans un processus de sérialisation et désérialisation.

- ◀ La date de naissance du client est mappée dans une colonne de type `DATE`.
- ◀ L'âge du client n'est pas stocké dans la base de données, cet attribut est `transient`.

ANNOTATIONS **Markup**

Nous avons vu le code de différentes annotations qui possèdent des méthodes. `@Embedded` et `@Embeddable` ne définissent aucune méthode. Elles sont appelées annotations markup.

Le bon de commande est stocké dans la table `t_order`.

Notez que nous pouvons ne pas spécifier la stratégie de génération de clé si on utilise la stratégie par défaut (`strategy = GenerationType.AUTO`).

La classe `Order` englobe les attributs de la classe `CreditCard` en utilisant l'annotation `@Embedded`.

La carte de crédit n'est pas annotée par `@Entity` mais par `@Embeddable`. Cela signifie que ses attributs se retrouvent dans la table de la classe englobante.

Attributs de la classe `Order`.

Englober deux objets dans une seule table

JPA permet d'englober les attributs de deux classes dans une seule table de manière relativement simple. Ceci est particulièrement utile lorsque les deux classes partagent le même cycle de vie (par exemple, lorsqu'on supprime un bon de commande, les lignes de commande n'ont plus de raison d'exister). La classe englobée utilise l'annotation `@Embeddable` alors que la classe englobante utilise `@Embedded`.

JPA permet d'englober les attributs de deux classes dans une seule table de manière relativement simple. La classe englobée utilise l'annotation `@Embeddable` alors que la classe englobante utilise `@Embedded`.

Prenons l'exemple d'un bon de commande (`Order`) que l'on règle à l'aide d'une carte bancaire (`CreditCard`). Ces deux classes peuvent être englobées dans une seule et même table (`t_order` dans notre exemple).

Entity `Order` englobant l'entity `CreditCard`

```
@Entity
@Table(name = "t_order")

public class Order {

    @Id
    @GeneratedValue
    private Long id;

    @Embedded
    private CreditCard creditCard;
    (...)
}
```

L'entity `CreditCard` est englobable

```
@Embeddable
public class CreditCard {
    private String creditCardNumber;
    private CreditCardType creditCardType;
    private String creditCardExpDate;
}
```

Remarquez que, sans ces annotations, les attributs de `Order` seraient stockés dans une table et les attributs de `CreditCard` dans une autre. Voici la DDL de la table des bons de commandes.

DDL de la table `t_order` contenant les attributs des deux entités

```
CREATE TABLE T_ORDER (
    ID BIGINT NOT NULL,
    ORDER_DATE DATE,
```

```
CREDIT_CARD_TYPE VARCHAR(255),  
CREDIT_CARD_NUMBER VARCHAR(30),  
CREDIT_CARD_EXPIRY_DATE VARCHAR(5),  
PRIMARY KEY (ID))  
)
```

- ◀ Attributs de la classe CreditCard.
- ◀ La clé primaire est celle de la classe Order.

Relations

Nous venons d’étudier toutes sortes d’annotations permettant de mapper une classe dans une table, un attribut dans une colonne et changer certaines valeurs par défaut. Le monde de l’orienté objet regorge aussi de relations entre classes (associations unidirectionnelles, bidirectionnelles, multiples, etc.). JPA permet de rendre persistante cette information de telle sorte qu’une classe peut être liée à une autre dans un modèle relationnel.

Il existe plusieurs types d’associations entre entities. Tout d’abord, une association possède un sens et peut être unidirectionnelle ou bidirectionnelle (c’est-à-dire qu’on peut naviguer d’un objet vers un autre et inversement). Ensuite, cette association possède une cardinalité, c’est-à-dire que nous pouvons avoir des liens 0:1, 1:1, 1:n ou n:m. Nous ne décrirons pas toutes les combinaisons possibles d’associations, mais juste celles utilisées dans l’application YAPS Pet Store.

Dans notre modèle, les classes sont reliées entre elles par des associations. Cette notion objet a son pendant dans le monde relationnel. JPA est donc capable de mapper des associations entre objets en relation entre tables.

Jointures

Dans le monde relationnel, il existe deux manières d’avoir une relation entre deux tables : en utilisant les clés étrangères ou les tables de jointures intermédiaires.

Par exemple, pour stocker le fait qu’un client possède une adresse, on peut soit avoir une clé étrangère dans la table du client, soit une table intermédiaire qui stockera cette information (figures 4–1 et 4–2).

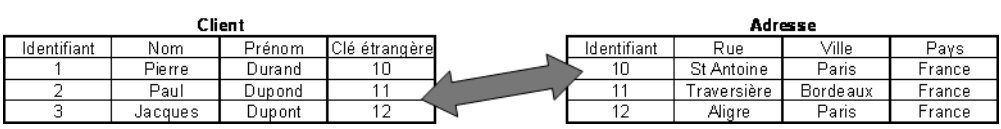
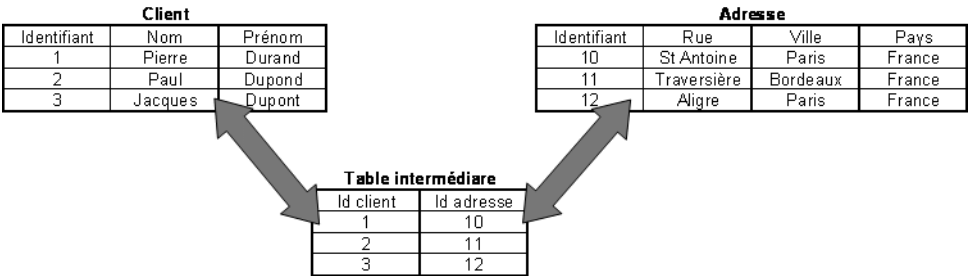


Figure 4–1
Utilisation de clés étrangères

Figure 4-2
Utilisation
d'une table intermédiaire



Comme nous le verrons dans les chapitres suivants, JPA utilise les deux modes de stockage. Par contre, JPA masque complètement cette implémentation puisque les classes, elles, utilisent le même type d'association dans les deux cas.

ANNOTATIONS **Programmation par exception**

Comme nous l'avons vu pour les annotations élémentaires, JPA utilise des valeurs et des paramètres par défaut (programmation par exception) pour effectuer son mapping. Ainsi, si un attribut n'est pas annoté, JPA utilisera les valeurs par défauts pour le rendre persistant. Il en est de même pour les relations. Il est possible de ne pas utiliser l'annotation `@OneToOne` si les paramètres par défaut nous satisfont.

► L'annotation `@OneToOne` définit un lien 1-1 entre le bon de commande et l'adresse de livraison.

► Pour respecter la cardinalité 1:1 et rendre la relation obligatoire, on utilise l'annotation `@JoinColumn`. On spécifie alors que la clé étrangère ne doit pas accepter la valeur `null` (`nullable=false`).

Relation unidirectionnelle 1:1

Une relation unidirectionnelle 1:1 entre classes est un lien de cardinalité 1 qui ne peut être accédé que dans un sens. Prenons l'exemple du bon de commande et de l'adresse de livraison. Les cas d'utilisation définissent qu'un bon de commande doit posséder une et une seule adresse de livraison (cardinalité 1). Il est important de naviguer du bon de commande vers l'adresse, par contre, il n'est pas utile de pouvoir naviguer dans le sens inverse. Le lien est donc unidirectionnel. JPA utilise l'annotation `@OneToOne` pour définir ce type de lien.

Relation unidirectionnelle 1:1 entre bon de commande et adresse

```
@Entity
@Table(name = "t_order")
public class Order {
    @Id @GeneratedValue
    private Long id;

    @OneToOne

    @JoinColumn(name = "address_fk", nullable = false)
    private Address deliveryAddress;
    (...)
}
```

Pour rendre un lien unidirectionnel, il suffit de ne pas avoir d'attribut `Order` dans la classe `Address`. Remarquez l'utilisation de l'annotation `@JoinColumn`. Celle-ci est utilisée pour paramétrer la colonne de la jointure (la clé étrangère). Dans notre exemple, on renomme la colonne en `address_fk` au lieu de `deliveryAddress` (qui serait son nom par défaut). On rend la relation obligatoire en refusant la valeur `null` dans cette colonne.

Code de l'annotation @javax.persistence.OneToOne

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OneToOne {
    Class targetEntity() default void.class;

    CascadeType[] cascade() default {};

    FetchType fetch() default FetchType.EAGER;

    boolean optional() default true;
    String mappedBy() default "";
}
```

- ◀ Nom de la classe de l'association. Dans notre exemple du bon de commande, targetEntity aurait pu être égale à la classe Address.
- ◀ Les opérations qui doivent être propagées à la classe associée (décrit par la suite dans ce chapitre).
- ◀ Chargement de la relation (décrit dans la suite de ce chapitre).
- ◀ L'association est facultative.
- ◀ Cet attribut n'est utilisé que dans un lien bidirectionnel.

Code de l'annotation @javax.persistence.JoinColumn

```
package javax.persistence;

@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface JoinColumn {
    String name() default "";
    String referencedColumnName() default "";

    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";

    String table() default "";
}
```

- ◀ Cette annotation s'applique à une classe, une méthode ou à un attribut.
- ◀ Nom de la colonne contenant la clé étrangère.
- ◀ Nom de la colonne référencée si ce n'est pas la clé primaire.
- ◀ La valeur doit-elle être unique ?
- ◀ La valeur null est-elle autorisée ?
- ◀ Peut-on avoir la colonne dans un ordre insert ou update ?
- ◀ Cet attribut peut contenir la définition de la colonne au format DDL.
- ◀ Utilisé lors d'un mapping multitable.

Voici le code DDL de la table t_order. La clé étrangère address_fk se trouve dans la table t_order et possède une contrainte d'intégrité référentielle.

DDL de la table t_order avec clé étrangère sur l'adresse

```
CREATE TABLE T_ORDER (
    ID BIGINT NOT NULL,
    ADDRESS_FK BIGINT NOT NULL
    PRIMARY KEY (ID)
)
ALTER TABLE T_ORDER
ADD CONSTRAINT T_ORDER_ADDRESS_FK
FOREIGN KEY (ADDRESS_FK) REFERENCES T_ADDRESS (ID);
```

- ◀ La colonne address_fk n'accepte pas la valeur null, le lien est donc obligatoire.
- ◀ À partir des annotations, JPA génère une contrainte d'intégrité référentielle entre la colonne address_fk de t_order et la clé primaire de la table t_address.

⚡ Contrainte d'intégrité

Permet de contraindre la modification des données d'une table, afin que les données saisies dans la base soient conformes aux données attendues. Dans le cas d'une clé étrangère, l'intégrité référentielle s'assure que la clé étrangère existe dans la table référencée (`add constraint`).

Relation unidirectionnelle 0:1

Une relation unidirectionnelle 0:1 est implémentée de la même manière qu'une relation 1:1. La seule différence réside dans le fait qu'elle est optionnelle. Prenons l'exemple d'un client et de son adresse. Dans les cas d'utilisation, il est décrit qu'un client n'est pas obligé de fournir son adresse au système. Par contre, si le client souhaite la saisir, il ne peut en avoir qu'une.

Pour transformer une relation 1:1 en 0:1, il suffit d'autoriser la valeur `null` dans la colonne. Pour cela, il est nécessaire de positionner l'attribut `nullable` de l'annotation `@JoinColumn` à `true`.

Relation unidirectionnelle 0:1 entre client et adresse

```
@Entity
@Table(name = "t_customer")
public class Customer {
    @Id @GeneratedValue
    private Long id;

    @OneToOne
    @JoinColumn(name = "address_fk", nullable = true)
    private Address homeAddress;
    (...)
}
```

Pour rendre la relation optionnelle (0:1), on autorise la valeur `null` (valeur par défaut que l'on aurait pu omettre).

DDL d'une relation 0:1 entre `t_customer` et `t_address`

```
CREATE TABLE T_CUSTOMER (
    ID BIGINT NOT NULL,
    ADDRESS_FK BIGINT
    PRIMARY KEY (ID)
)

ALTER TABLE T_CUSTOMER
ADD CONSTRAINT T_CUSTOMER_ADDRESS_FK
FOREIGN KEY (ADDRESS_FK) REFERENCES T_ADDRESS (ID)}
```

La colonne `address_fk` accepte la valeur `null`, le lien est donc optionnel.

À partir des annotations, JPA génère une contrainte d'intégrité référentielle entre la colonne `address_fk` de la table `t_customer`, et la clé primaire de la table `t_address`.

Relation bidirectionnelle 1:n

Une relation 1:n signifie qu'un objet fait référence à un ensemble d'autres objets (cardinalité n). Dans l'application YAPS Pet Store, cette notion est décrite dans le catalogue, par exemple, où une catégorie con-

tient plusieurs produits. De plus, elle est bidirectionnelle puisque le produit a connaissance de la catégorie à laquelle il appartient. Cette information de cardinalité en Java est décrite par les structures du paquetage `java.util` : `Collection`, `List`, `Map` et `Set`. JPA utilise les annotations `@OneToMany` et `@ManyToOne`.

Les collections en Java

Les collections (paquetage `java.util`) proposent une série de classes, d'interfaces et d'implémentations pour gérer les structures de données (listes, ensembles). Chaque implémentation utilise une stratégie avec des avantages et des inconvénients : certaines collections acceptent les doublons, d'autres non ; certaines sont ordonnées, d'autres pas.

Les `java.util.Set` (ensembles) sont un groupe d'éléments uniques.

Les `java.util.List` (listes) sont une suite d'éléments ordonnés accessibles par leur rang dans la liste. Les listes ne garantissent pas l'unicité des éléments.

Les `java.util.Map` mémorisent une collection de couples clé-valeur. Les clés sont uniques, mais la même valeur peut-être associée à plusieurs clés.

Une catégorie possède une liste de produits

```
@Entity
@Table(name = "t_category")
public class Category
{
    (...)

    @OneToMany(mappedBy = "category")
    private List<Product> products;
    (...)
}
```

◀ Les données de l'entity `Category` sont rendues persistantes dans la table `t_category`.

◀ Une (One) catégorie possède plusieurs (Many) produits. Remarquez l'utilisation des génériques pour la liste de produits.

Le produit a connaissance de sa catégorie

```
@Entity
@Table(name = "t_product")
public class Product
{
    (...)

    @ManyToOne
    @JoinColumn(name = "category_fk")
    private Category category;
    (...)
}
```

◀ Les données de l'entity `Product` sont rendues persistantes dans la table `t_product`.

◀ Plusieurs (Many) produits peuvent être rattachés à une (One) même catégorie. On renomme la colonne de la clé étrangère en `category_fk`.

Dans une relation bidirectionnelle, JPA peut utiliser deux modes de jointure : le système de clé étrangère ou la table de jointure. Si aucune annotation n'est utilisée, la table de jointure est le mode par défaut. On se retrouve alors avec une table (`t_category_product` par exemple) contenant deux colonnes permettant de stocker la relation entre catégorie et produit.

La table `t_category` ne porte pas l'information du lien avec `t_product`.

La table `t_product` utilise une clé étrangère pour pointer vers la table `t_category`.

Contrainte d'intégrité référentielle sur la clé étrangère `category_fk`.

Si ce mode par défaut n'est pas satisfaisant, il faut utiliser l'attribut `mappedBy` de l'annotation `@OneToMany`. Dans notre exemple, le fait que l'entity `Category` déclare `@OneToMany(mappedBy = "category")` précise à JPA qu'il doit utiliser le système de clé étrangère. L'attribut `mappedBy` n'a de sens que dans une relation bidirectionnelle.

DDL des tables `t_category` et `t_product`

```
CREATE TABLE T_CATEGORY (
    ID BIGINT NOT NULL,
    PRIMARY KEY (ID)
)

CREATE TABLE T_PRODUCT (
    ID BIGINT NOT NULL,
    CATEGORY_FK BIGINT,
    PRIMARY KEY (ID)
)

ALTER TABLE T_PRODUCT
    ADD CONSTRAINT T_PRODUCT_CATEGORY_FK
    FOREIGN KEY (CATEGORY_FK) REFERENCES t_category (ID)
```

Association multiple sans générique

L'entity `Category` utilise les génériques pour la liste des produits. Grâce aux génériques qui typent cette liste, JPA sait que la persistance doit se faire entre `Category` et `Product`. Si vous n'utilisez pas les génériques, JPA ne saura pas sur quel autre entity pointer. Il faut alors spécifier la classe de l'entity dans la relation à l'aide de l'attribut `targetEntity` de l'annotation `@OneToMany`.

```
@Entity
@Table(name = "t_category")
public class Category
{
    (...)
    @OneToMany(mappedBy = "category", targetEntity = Product.class)
    private List products;
    (...)
}
```

Ci-après le code des deux annotations utilisées pour la cardinalité 1:n.

Code de l'annotation `@javax.persistence.OneToMany`

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public interface OneToMany {
    Class targetEntity() default void.class;
}
```

Définit l'entity avec lequel il faut créer une relation. Cet attribut est obligatoire lorsqu'on n'utilise pas les types génériques.

```

CascadeType[] cascade() default {};

FetchType fetch() default FetchType.LAZY;

}

```

- ◀ Opérations devant être propagées à la classe associée (décrit dans la suite de ce chapitre).
- ◀ Chargement de la relation (décrit par la suite dans ce chapitre).
- ◀ Utilisé lorsqu'on veut créer un lien avec une clé étrangère plutôt qu'une table de jointure.

Code de l'annotation @javax.persistence.ManyToOne

```

package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface ManyToOne {
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};

    FetchType fetch() default FetchType.EAGER;

    boolean optional() default true;
}

```

- ◀ Définit l'entity avec lequel il faut créer une relation. Cet attribut est obligatoire lorsqu'on n'utilise pas les types génériques.
- ◀ Opérations devant être propagées à la classe associée (décrit dans la suite de ce chapitre).
- ◀ Chargement de la relation (décrit par la suite dans ce chapitre).
- ◀ L'association est-elle facultative ?

Relation unidirectionnelle 1:n

Les relations unidirectionnelles 1:n sont particulières. En effet, JPA ne permet pas l'utilisation des clés étrangères pour mapper cette relation mais uniquement le système de table de jointure. Par défaut, le nom de cette table intermédiaire est composé du nom des deux entités séparés par le caractère '_'. Une fois de plus, les annotations JPA permettent de redéfinir ces valeurs par défaut (en utilisant @JoinTable).

Prenons l'exemple d'un bon de commande. Un bon de commande (Order) est composé de plusieurs lignes de commande (OrderLine). La relation est donc multiple et la navigation se fait uniquement dans le sens Order vers OrderLine. Ci-après, le code de l'entity Order avec une relation unidirectionnelle 1:n vers OrderLine.

Entity Order avec l'annotation @JoinTable

```

@Entity
@Table(name = "t_order")
public class Order
{
    ...
    @OneToMany
    @JoinTable(name = "t_order_order_line",
        joinColumns = {@JoinColumn(name = "order_fk")},
        inverseJoinColumns = {@JoinColumn(name = "order_line_fk")})
    private List<OrderLine> orderLines;
    ...
}

```

- ◀ Les données de l'entity Order sont rendues persistantes dans la table t_order.
- ◀ La table de jointure est renommée t_order_order_line ainsi que les colonnes des clés étrangères.

Table `t_order` contenant les données du bon de commande.

La table des lignes de commande ne possède pas de clé étrangère vers le bon de commande.

Cette table crée la jointure entre le bon de commande et ses lignes de commande. La clé primaire de cette table est constituée des deux clés étrangères.

Contraintes d'intégrité référentielle sur la table de jointure.

PERSISTANCE Nom des contraintes d'intégrité

Le nom des contraintes d'intégrité est généré automatiquement par JPA. Parfois, ce nom peut être assez exotique, comme pour les contraintes de la table intermédiaire entre bons de commandes qui se nomment `TRDRORDERLINERDRFK` et `TRDRRDRLINERDRLNFK`.

Nom de la table de jointure.

Catalogue et schéma de la table.

Colonne de la clé étrangère faisant référence à la clé primaire de la table qui maintient la relation.

Colonne de la clé étrangère faisant référence à la clé primaire de la seconde table.

Ce tableau permet de définir les contraintes d'unicité sur une ou plusieurs colonnes.

Le code précédent indique à JPA de créer une table pour les bons de commande et une autre pour faire la jointure avec les lignes de commandes. Voici la DDL que l'on obtient.

DDL de relation unidirectionnelle avec table de jointure

```
CREATE TABLE T_ORDER (
  ID BIGINT NOT NULL,
  PRIMARY KEY (ID)
)

CREATE TABLE T_ORDER_LINE (
  ID BIGINT NOT NULL,
  PRIMARY KEY (ID)
)

CREATE TABLE T_ORDER_ORDER_LINE (
  ORDER_FK BIGINT NOT NULL,
  ORDER_LINE_FK BIGINT NOT NULL,
  PRIMARY KEY (ORDER_FK, ORDER_LINE_FK)
)

ALTER TABLE T_ORDER_ORDER_LINE
  ADD CONSTRAINT TRDRORDERLINERDRFK
  FOREIGN KEY (ORDER_FK) REFERENCES T_ORDER (ID)

ALTER TABLE T_ORDER_ORDER_LINE
  ADD CONSTRAINT TRDRRDRLINERDRLNFK
  FOREIGN KEY (ORDER_LINE_FK) REFERENCES T_ORDER_LINE (ID)
```

L'annotation `@JoinTable` est utilisée lorsqu'on veut redéfinir les valeurs par défaut de la table de jointure.

Code de l'annotation `@javax.persistence.JoinTable`

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface JoinTable {

  String name() default "";

  String catalog() default "";
  String schema() default "";

  JoinColumn[] joinColumns() default {};

  JoinColumn[] inverseJoinColumns() default {};

  UniqueConstraint[] uniqueConstraints() default {};
}
```

Relation n:m

L'étude de cas ne comporte pas de relations n:m mais cet aparté vous en explique le fonctionnement. Prenons par exemple un article (Item) qui peut avoir plusieurs étiquettes (Label) et vice versa.

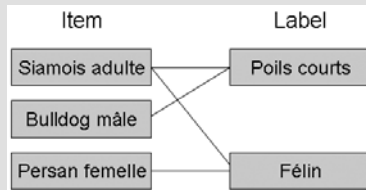


Figure 4–3 Relation n:m entre articles (Item) et étiquettes (Label)

Pour les relations n:m (qu'elles soient unidirectionnelles ou bidirectionnelles), il faut utiliser l'annotation `@ManyToMany` dans les deux entités. Dans le cas d'une relation bidirectionnelle il faut rajouter l'attribut `mappedBy`. Nous avons donc la classe Item qui définit une liste de Label annotée avec `@ManyToMany`.

```
@Entity
public class Item {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
```

```
private String name;
    @ManyToMany(mappedBy = "item")
    private List<Label> labels;
}
```

Dans le sens inverse, la classe Label utilise elle aussi une annotation `@ManyToMany` sur la liste d'articles.

```
@Entity
public class Label {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String name;
    @ManyToMany
    private List<Item> items;
}
```

Le résultat est que chaque entity persistera ses données dans une table et la relation n:m sera stockée dans une table de jointure. Par défaut, le nom de cette table intermédiaire est composé des noms des deux entités séparés par le caractère '_' (l'annotation `@JoinTable` permet de redéfinir cette valeur).

- <http://www.devx.com/Java/Article/33650/>
- <http://www.devx.com/Java/Article/33906>

Chargement d'une association

Toutes les annotations de relations que nous venons de voir (`@OneToOne`, `@OneToMany`, `@ManyToOne`) définissent un attribut agissant sur le chargement des relations. L'attribut `fetch` permet de spécifier si vous souhaitez que les objets associés se chargent directement (EAGER) ou de manière différée (LAZY).

Dans le cas d'un mode de chargement EAGER, les objets liés sont automatiquement chargés. Dans le cas du mode LAZY, les objets sont chargés uniquement lorsqu'on y accède explicitement. Prenons l'exemple de la catégorie, qui est liée à une liste de produits, et un produit lié à sa catégorie (lien bidirectionnel). Lorsqu'on charge une catégorie, on ne veut pas que la liste des produits se charge automatiquement. On souhaite pouvoir accéder à cette liste seulement lorsque l'on appelle l'accessor `category.getProducts()`, c'est-à-dire de manière différée (LAZY). À l'inverse, lorsqu'on charge un produit, on veut que sa catégorie soit automatiquement chargée (EAGER).

Le paramètre `fetch` est très important car il peut provoquer des problèmes de performances s'il est mal utilisé. Imaginez un modèle objet riche et complexe, où toutes les relations sont définies automatiquement (EAGER). Cela signifierait qu'à l'appel d'un objet, le système aurait à charger automatiquement toute la grappe d'objets liés. Cela aurait des impacts sur la mémoire du système ainsi que sur les performances de la base de données.

PRÉCISION Mode fetch par défaut

Selon le type de relation, l'attribut `fetch` est par défaut soit en mode EAGER soit en mode LAZY :

Relation	Fetch par défaut
@Basic	EAGER
@OneToOne	EAGER
@ManyToOne	EAGER
@OneToMany	LAZY
@ManyToMany	LAZY

L'entityCategory utilise un chargement différé pour charger la relation vers ses produits.

Le produit déclare le chargement automatique de sa catégorie.

Une catégorie possède une liste de produits

```
@Entity
public class Category
{
    @OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
    private List<Product> products;
    (...)
}

@Entity
public class Product
{
    @ManyToOne (fetch = FetchType.EAGER)
    private Category category;
    (...)
}
```

Ordonner une association multiple

Les bases de données relationnelles ne préservent pas d'ordre dans leur table. Ainsi, si on veut récupérer une liste ordonnée de telle ou telle manière, il faut utiliser le mot-clé `order by` dans les ordres SQL. Il en va de même pour les listes d'entities.

Pour reprendre l'exemple de la catégorie et de ses produits, on veut pouvoir récupérer cette liste de manière ordonnée (par ordre ascendant des noms de produits). Pour cela, JPA propose l'annotation `@OrderBy` que l'on peut utiliser sur les annotations `@OneToMany` et `@ManyToOne`. `@OrderBy` prend en paramètre les noms des attributs sur lesquels on souhaite effectuer un tri, ainsi que le mode (ascendant ou descendant).

Code de l'annotation `@javax.persistence.OrderBy`

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OrderBy {
    String value() default "";
}
```

Chaîne de caractères contenant les noms des attributs de l'entity sur lesquels effectuer le tri.

Ainsi, pour classer la liste des produits dans l'ordre ascendant du nom, on utilise la chaîne de caractères " name ASC " ou " name DESC " pour l'ordre descendant. On peut aussi utiliser plusieurs attributs comme " name ASC, description DESC " pour trier dans l'ordre croissant des noms et décroissant de la description.

Les produits d'une catégorie sont classés dans l'ordre ascendant du nom

```
@Entity
public class Category
    @OneToMany(mappedBy = "category", fetch = FetchType.LAZY)
    @OrderBy("name ASC")
    private List<Product> products;
    (...)
}

@Entity
public class Product
    private String name;
    @ManyToOne (fetch = FetchType.EAGER)
    private Category category;
    (...)
}
```

◀ Lorsqu'on accède aux produits de la catégorie, la liste est ordonnée en mode ascendant sur le nom des produits (name ASC).

◀ L'entity produit a un attribut nom (name) sur lequel l'ordonnement se fait.

Cascade

Parfois, lorsqu'on effectue une opération sur un entity, on souhaite que celle-ci se propage sur les associations. On parle alors d'action en cascade. Par exemple, lorsqu'on supprime une catégorie du système, on veut que ses produits soient également supprimés.

La catégorie supprime ses produits en cascade

```
@Entity
public class Category
    @OneToMany(cascade = CascadeType.REMOVE)
    private List<Product> products;
    (...)
}
```

◀ L'attribut cascade est présent dans les annotations que nous avons vu précédemment : @ManyToOne, @OneToMany, et @OneToOne

Héritage

L'héritage est une notion intégrée dans les langages objets en général et dans Java en particulier. Mais le stockage de cette information hiérarchique dans une structure relationnelle n'est pas facile à réaliser. La preuve en est que, jusqu'à l'apparition de JPA, l'héritage n'était pas possible en J2EE (entity bean 2.x). Grâce à JPA, le polymorphisme et l'héritage sont maintenant possibles.

Contrairement aux associations qui ont un mapping assez simple (système de clé étrangère ou de table de jointure), l'héritage demande au développeur de choisir entre plusieurs stratégies de mapping :

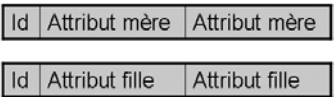
- **SINGLE_TABLE** : une seule table par hiérarchie de classes (stratégie par défaut). Cela signifie que les attributs de la classe mère et ceux des classes filles sont « aplatis » dans une seule et même table (figure 4-4).

PRÉCISION

Pas d'héritage dans le YAPS Petstore

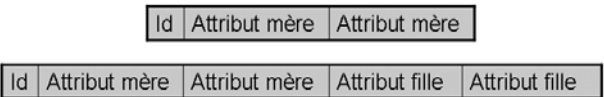
L'application YAPS Petstore ne présente pas de cas d'héritage. Les notions sur l'héritage JPA sont présentées ici sans base d'exemples concrets de l'application.

Figure 4-4
Stratégie Single Table



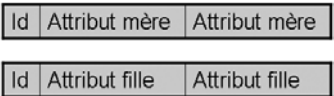
- TABLE_PER_CLASS : une table par classe concrète. Les attributs des classes filles, y compris les propriétés héritées, sont mappés dans une table. Les attributs de la classe mère sont stockés dans une table à part pour effectuer les jointures. Notez que cette stratégie est facultative dans la spécification JPA 1.0 (figure 4-5).

Figure 4-5
Stratégie Table Per Class



- JOINED : une table par classe. Dans cette stratégie, la classe mère ainsi que les classes filles sont mappées dans une table. Les attributs hérités de la table mère ne se retrouvent pas dans les tables filles (figure 4-6).

Figure 4-6
Stratégie Joined



Le choix de la stratégie s'effectue dans la classe mère en utilisant l'annotation `@javax.persistence.Inheritance` conjointement au type d'héritage (`SINGLE_TABLE`, `TABLE_PER_CLASS` ou `JOINED`).

Classe mère

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Address {

    @Id
    private Long id;
    private String street1;
    // (...)
}
```

Classe fille

```
@Entity
public class CompanyAddress extends Address {

    private String postmail;
    // (...)
}
```

Le cycle de vie d'un entity

Comme nous l'avons déjà dit, avec l'arrivée de JPA, un entity est maintenant une simple classe Java. Un simple appel au mot-clé `new` permet d'instancier un entity et de le manipuler comme toute autre classe. La seule différence notable est l'utilisation des annotations que nous venons de voir permettant à JPA de prendre en compte cette classe et de la rendre persistante. On dit alors que la classe est « *managée* » par JPA. Le traitement inverse, lorsque la classe cesse d'être « *managée* », est appelé « *détaché* ».

Pour illustrer ces principes, prenons une classe persistante, `Category` par exemple. Lorsque cette classe veut accéder à la base de données, elle a besoin d'être *managée*. Par contre, lorsqu'elle doit traverser plusieurs couches de l'application pour être affichée sur la partie cliente, elle se détache et ne peut plus alors manipuler les données. Un entity cesse automatiquement d'être « *managé* » lorsqu'il se sérialise.

Le diagramme d'état suivant décrit les différents états que peut prendre un entity. Il doit être lu comme suit. Lorsqu'on instancie un entity, cet objet existe en mémoire uniquement. Ce n'est que lorsque JPA le prend en compte qu'il devient *managé*. Si on met à jour l'entity, ou que l'on parcourt ses relations vers d'autres objets, il reste *managé*. Il se détache lorsqu'il se « *déplace* » vers une autre couche, par exemple, et peut être rattaché (donc *managé*) à son retour. Lorsqu'on supprime un entity, il supprime ses données de la base avant de disparaître de la mémoire de la JVM.

Notez la présence de déclencheurs sur chaque transition (`@PrePersist`, `@PreRemove`, etc.). Ces points d'attache, ou méthodes de callback, sont appelés par JPA lorsque l'entity change d'état.

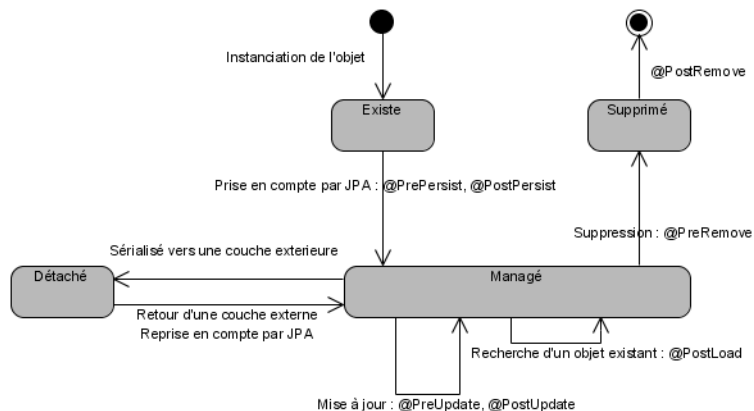


Figure 4–7
Cycle de vie d'un entity

UML Diagramme d'états

Ce diagramme sert à représenter des automates d'états finis, sous forme de graphes d'états, reliés par des arcs orientés qui décrivent les transitions. Les diagrammes d'états permettent de décrire les changements d'états d'un objet ou d'un composant, en réponse aux interactions avec d'autres objets ou avec des acteurs.

SYNTAXE Les méthodes callback

Une méthode callback possède les caractéristiques suivantes :

- elle peut être annotée par une ou plusieurs annotations callback ;
- elle ne peut lancer que des `RuntimeException` ;
- elle peut être `public`, `package`, `protected` ou `private`.

REMARQUE Exceptions

Lorsque la valeur d'un attribut est invalide, les entités lèvent une `ValidationException`. Cette exception applicative sera vue en détail dans le prochain chapitre, *Traitements métier*.

► L'entity `Category` a un attribut `nom` et `description`.

► Avant d'effectuer une insertion ou une mise à jour des données en base, JPA appelle la méthode `validateData`. Cette méthode s'assure que les attributs `nom` et `description` sont valides.

Les annotations de callback

Grâce aux annotations de callback, JPA laisse la possibilité aux développeurs de rajouter des traitements avant ou après qu'un changement d'état se produise. Il existe sept annotations qui correspondent à ces différents points d'attache.

- `@javax.persistence.PrePersist`
- `@javax.persistence.PostPersist`
- `@javax.persistence.PreRemove`
- `@javax.persistence.PostRemove`
- `@javax.persistence.PreUpdate`
- `@javax.persistence.PostUpdate`
- `@javax.persistence.PostLoad`

Avant d'insérer un entity en base, JPA exécute les méthodes annotées par `@PrePersist`. Si l'insertion ne rencontre pas de problèmes ou d'exceptions, les méthodes annotées `@PostPersist` sont exécutées. Il en est de même pour les mises à jour (`@PreUpdate`, `@PostUpdate`) et les suppressions (`@PreRemove`, `@PostRemove`). Par contre, l'annotation `@PostLoad` est appelée lorsqu'un entity est chargé à partir de la base de données via une requête ou une association.

Dans l'exemple suivant, l'entity `Category` valide ses données avant son insertion en base (`@PrePersist`) ou sa mise à jour (`@PreUpdate`). Si les données ne sont pas valides, une exception est lancée.

Catégorie utilisant des annotations de callback

```
@Entity
public class Category
{
    @Id
    @GeneratedValue
    private Long id;

    private String name;
    private String description;
    (...)

    @PrePersist
    @PreUpdate
    private void validateData() {
        if (name == null || "".equals(name))
            throw new ValidationException("Invalid name");
        if (description == null || "".equals(description))
            throw new ValidationException("Invalid description");
    }
}
```

Les entités de YAPS Pet Store

Après avoir découvert le fonctionnement de JPA et de certaines annotations, il ne nous reste plus qu'à étudier son application sur les objets persistants de YAPS Pet Store.

Pour cela, il nous faut définir les objets persistants de l'application à partir des cas d'utilisation du premier chapitre. Un moyen simple consiste à trouver les noms (et pas les verbes) qui reviennent fréquemment dans les cas d'utilisation. Les objets à rendre persistants représentent souvent des choses, comme un article ou un bon de commande. Ils encapsulent leurs données (on parle d'attributs) et leur comportement (les méthodes).

Charge à l'analyste qui sommeille en nous de trouver les objets persistants. Ainsi, il n'y aura pas d'objet catalogue (la société YAPS n'a qu'un seul catalogue) mais plutôt des catégories (Category), des produits (Product) et des articles (Item). On retrouvera aussi des clients (Customer) et des adresses (Address). Lorsque les achats sont effectués, on obtient un bon de commande (Order), constitué de lignes de commande (OrderLine) et payable par carte bancaire (CreditCard).

RETOUR D'EXPÉRIENCE **Anglais ou français ?**

L'informatique, et plus précisément le langage Java, est dominée par la langue anglaise. À tel point que certaines spécifications nous imposent des termes anglo-saxons (par exemple, les Java beans doivent avoir des accesseurs qui commencent par les mots `get` et `set`, ou une méthode `toString`). Ces règles nous imposant l'anglais, il nous faut ensuite mélanger le français pour obtenir un franglais que je trouve plus difficile à lire (ex. `getNom`, `setPrenom` alors que `getName` ou `setAddress` sont plus clairs et compris de tous). La plupart des développeurs sont maintenant habitués à utiliser des termes anglais pour les classes, méthodes et attributs. Il est sage de l'imposer comme règle de développement dans une équipe pour éviter le franglais ou le mélange des deux langues dans le code source. Ce livre utilisera donc uniquement des termes anglais pour les classes, attributs et méthodes. Par contre, les commentaires seront en français.

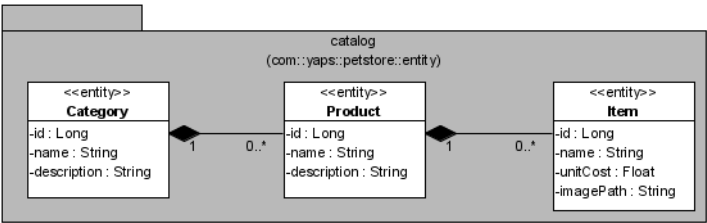
Le catalogue

La société YAPS possède un catalogue d'animaux domestiques. Ce catalogue est structuré en catégories, en produits puis en articles. Ce sont ces articles que les clients peuvent acheter. Ci-après le diagramme de classes représentant ce découpage (figure 4–8).

UML Les mots dans les cas d'utilisation

Cette méthode simple, consistant à trouver les noms qui reviennent fréquemment dans les cas d'utilisation, est tellement répandue que certains outils l'utilisent. Visual Paradigm, par exemple, possède un système d'analyse textuelle (*Textual Analysis*) qui, à partir du texte, réalise des diagrammes de classes approximatifs.

Figure 4–8
Diagramme de classes du catalogue



UML Diagramme de classes

Un diagramme de classes est une collection d'éléments de modélisation statiques (classes, interfaces, paquetages, etc.), qui montrent la structure d'un modèle. Il fait abstraction des aspects dynamiques et temporels. Ce diagramme se concentre sur les relations entre classes (association, héritage, etc.), leurs attributs et méthodes.

On doit lire le diagramme de la manière suivante. Une catégorie (Category) peut avoir zéro ou plusieurs produits (Product), et un produit peut avoir zéro ou plusieurs articles (Item). On notera l'utilisation de liens de composition bidirectionnels entre ces classes. La composition (le losange noir à l'extrémité de la classe) nous indique qu'il y a une relation forte entre ces objets. Ainsi, la suppression d'une catégorie entraînera la suppression de ses produits et des articles liés à ce produit (JPA utilise la notion de cascade).

En ce qui concerne les attributs de ces trois classes, ils suivent l'expression des besoins décrite dans le premier chapitre. La catégorie et le produit comportent un identifiant (id), un nom (name) et une description (description). L'article possède un prix unitaire (unitCost) et une image représentant l'animal à vendre (imagePath).

PERSISTANCE Les images

Pour pouvoir afficher une image pour chaque animal, il est nécessaire que cette image soit stockée dans le système. Deux possibilités s'offrent à nous. Soit l'image est stockée directement en base de données sous forme de BLOB (*Binary Large Object*), soit le nom du répertoire physique où se trouve l'image est stocké en base de donnée. Cette dernière solution est plus simple à mettre en œuvre. L'attribut imagePath de la classe Item contient donc le chemin d'accès à l'image (par exemple : /images/poissonRouge.gif).

Les relations dans un diagramme de classes UML

Dans un diagramme de classes, il existe quatre types de relations :

- **Héritage** : mécanisme par lequel des éléments plus spécifiques incorporent la structure et le comportement d'éléments plus généraux. En UML, on représente un héritage par une ligne avec un triangle à son extrémité.
- **Association** : relation sémantique entre deux ou plusieurs classes. C'est une connexion, bidirectionnelle par défaut, entre leurs instances. Représentée par une ligne.
- **Agrégation** : une forme spéciale d'association qui spécifie une relation « tout-partie » entre l'agrégat (le tout) et une partie. Représentée par une ligne avec un losange vide à une extrémité.
- **Composition** : une forme d'agrégation qui exprime une forte propriété entre le tout et les parties, ainsi qu'une subordination entre l'existence des parties et du tout. Les parties vivent et meurent avec le tout (elles partagent sa durée de vie). La composition est représentée par une ligne avec un losange plein (noir) à une extrémité.

Catégorie

La société YAPS vend cinq catégories d'animaux domestiques : chats, chiens, reptiles, poissons, oiseaux. Chaque catégorie contient une liste de

produits ❶ triée par nom ❷, à laquelle on accède de manière différée ❸. L'utilisation des annotations de callback permet à l'entity de vérifier la validité de ses attributs avant l'insertion et la mise à jour en base de données ❹.

Code de l'entity Category

```
package com.yaps.petstore.entity.catalog;

@Entity
@Table(name = "t_category")
public class Category implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false, length = 30)
    private String name;
    @Column(nullable = false)
    private String description;

    @OneToMany(mappedBy = "category",
                cascade = CascadeType.REMOVE,
                fetch = FetchType.LAZY) ❸
    @OrderBy("name ASC") ❷
    private List<Product> products; ❶

    @PrePersist
    @PreUpdate
    private void validateData() { ❹
        if (name == null || "".equals(name))
            throw new ValidationException("Invalid name");
        if (description == null || "".equals(description))
            throw new ValidationException("Invalid description");
    }

    // constructeurs, accesseurs
    // méthodes hashCode, equals et toString
}
```

❶ Entity Category rendant persistantes ses données dans la table t_category.

❷ Une catégorie a un identifiant unique, un nom et une description.

❸ Une catégorie possède une liste de produits accessible de manière différée. La suppression d'une catégorie entraîne la suppression en cascade de tous ses produits. La liste est triée par nom de produits.

❹ Avant d'insérer ou de mettre à jour les données en base, cette méthode est appelée par JPA. Elle permet de valider les attributs de l'entity et renvoie une exception en cas d'incohérence.

REMARQUE Implémenter Serializable

Notez que l'entity Category implémente l'interface `Serializable`. Comme nous l'avons vu dans notre architecture, les entités vont être utilisées par les applications clientes (Swing et JSP). Ils vont donc être transportés à travers le réseau. Java utilise l'interface `Serializable` ou `Externalizable` pour pouvoir transporter des objets à travers le réseau. On appelle cela la sérialisation des données. La plupart de nos entités seront utilisés par des applications distantes, ils implémenteront donc l'interface `Serializable`.

Produit

Chaque catégorie d'animaux domestiques est subdivisée en produits. Ainsi, la catégorie chats contiendra les produits Siamois, Persan, Chartreux, etc. L'entity Product possède un lien bidirectionnel avec la catégorie, il a donc un attribut qui lui fait référence ❶. Un produit contient une liste d'articles ❷, triée par nom ❸, et accédée de manière différée ❹.

Code de l'entity Product

```
package com.yaps.petstore.entity.catalog;

@Entity
```

Entity **Product** rendant persistantes ses données dans la table `t_product`.

Un produit a un identifiant unique, un nom et une description.

Le produit a connaissance de sa catégorie. Cette association est chargée automatiquement.

Un produit possède une liste d'articles accessible de manière différée. La suppression d'un produit entraîne la suppression en cascade de tous ses articles. La liste est triée par nom d'articles.

Avant d'insérer ou de mettre à jour les données en base, cette méthode est appelée par JPA. Elle permet de valider les attributs de l'entity et renvoie une exception en cas d'incohérence.

```
@Table(name = "t_product")
public class Product implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false, length = 30)
    private String name;
    @Column(nullable = false)
    private String description;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "category_fk", nullable = false)
    private Category category; ❶

    @OneToMany(mappedBy = "product",
                cascade = CascadeType.REMOVE,
                fetch = FetchType.LAZY) ❷
    @OrderBy("name ASC") ❸
    private List<Item> items; ❹

    @PrePersist
    @PreUpdate
    private void validateData() {
        if (name == null || "".equals(name))
            throw new ValidationException("Invalid name");
        if (description == null || "".equals(description))
            throw new ValidationException("Invalid description");
    }

    // constructeurs, accesseurs
    // méthodes hashCode, equals et toString
}
```

Article

Chaque produit est subdivisé en articles. Le produit chat Siamois comporte donc les articles suivants : adulte mâle et adulte femelle. L'article est l'élément qu'il est possible d'acheter dans le catalogue de la société YAPS. Un client peut donc rajouter ces articles dans son panier et les acheter. Un article a un prix unitaire et possède un lien bidirectionnel avec son produit ❶.

Code de l'entity Item

```
package com.yaps.petstore.entity.catalog;

@Entity
@Table(name = "t_item")
public class Item implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false, length = 30)
    private String name;
```

Entity **Item** rendant persistantes ses données dans la table `t_item`.

Un article a un identifiant unique, un nom, un prix unitaire et une image représentant l'animal domestique.

```

@Column(name = "unit_cost", nullable = false)
private Float unitCost;
@Column(name = "image_path")
private String imagePath;

@ManyToOne(fetch = FetchType.EAGER)
@JoinColumn(name = "product_fk", nullable = false)
private Product product; ❶

@PrePersist
@PreUpdate
private void validateData() {
    if (name == null || "".equals(name))
        throw new ValidationException("Invalid name");
}

// constructeurs, accesseurs
// méthodes hashCode, equals et toString
}

```

- ❶ L'article a connaissance de son produit. Cette association est chargée automatiquement.
- ❷ Avant d'insérer ou de mettre à jour les données en base, cette méthode est appelée. Elle permet de valider les attributs de l'entity et renvoie une exception en cas d'incohérence.

Le client

Les cas d'utilisation nous ont donné plusieurs informations sur la gestion des clients, ce qui nous permet d'extraire le diagramme de classes suivant (figure 4–9).

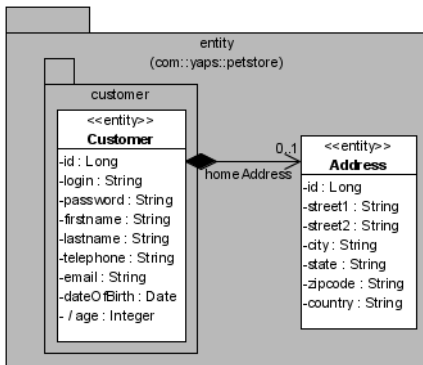


Figure 4–9
Diagramme de classes du client

Un client possède un identifiant (`id`) et un login unique. Grâce à ce login et à son mot de passe (`password`), il peut se connecter au système. Ses coordonnées sont constituées d'un nom (`lastname`), prénom (`firstname`), numéro de téléphone (`telephone`), adresse e-mail (`email`), date de naissance (`dateOfBirth`) et une adresse personnelle (optionnelle d'où la cardinalité 0:1). Cette adresse est décrite par la classe `Address` et comporte les attributs rue (`Street1`, `Street2`), ville (`city`), état (`state`), code postal (`zipcode`) et pays de résidence (`country`).

UML. Attribut dérivé

Remarquez le caractère « / » devant l'attribut âge. En UML, il signifie que l'élément (attribut ou méthode) est dérivé, c'est-à-dire qu'il est calculé à partir d'autres attributs. Dans le cas du client, l'âge est calculé à partir de sa date de naissance.

Client

L'entity **Customer** (client) est une classe riche en informations. Outre ses attributs, cet entity utilise plusieurs annotations de callback pour valider ses données ❶ mais aussi pour calculer l'âge du client ❷. Il utilise une méthode métier ❸ pour vérifier que le mot de passe, saisi par le client lors de la connexion, est bien le même que celui stocké en base de données. Cette méthode métier est appellable par des composants externes (la méthode est publique) et n'utilise pas d'annotations de callback. Un client possède un lien unidirectionnel avec l'entity **Address** ❹.

Code de l'entity **Customer**

Entity **Customer** rendant persistantes ses données dans la table `t_customer`.

Un client possède un identifiant unique, un nom, un prénom, un numéro de téléphone et une adresse e-mail. Le login et le mot de passe sont utilisés pour se connecter au système.

À partir de la date de naissance, on calcule l'âge du client (dans la méthode `calculateAge`). Cet attribut n'est pas stocké en base de données (`transient`).

Association unidirectionnelle vers l'adresse, chargée automatiquement.

Avant d'insérer ou de mettre à jour les données en base, cette méthode est appelée. Elle permet de valider les attributs de l'entity et renvoie une exception en cas d'incohérence.

```
package com.yaps.petstore.entity.customer;

@Entity
@Table(name = "t_customer")
public class Customer implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(unique = true, nullable = false, length = 8)
    private String login;
    @Column(nullable = false, length = 8)
    private String password;
    @Column(nullable = false, length = 30)
    private String firstname;
    @Column(nullable = false, length = 30)
    private String lastname;
    private String telephone;
    private String email;

    @Column(name = "date_of_birth")
    @Temporal(TemporalType.DATE)
    private Date dateOfBirth;
    @Transient
    private Integer age;

    @OneToOne(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name = "address_fk", nullable = true)
    private Address homeAddress; ❹

    @PrePersist
    @PreUpdate
    private void validateData() { ❶
        if (firstname == null || "".equals(firstname))
            throw new ValidationException("Invalid first name");
        if (lastname == null || "".equals(lastname))
            throw new ValidationException("Invalid last name");
        if (login == null || "".equals(login))
            throw new ValidationException("Invalid login");
        if (password == null || "".equals(password))
            throw new ValidationException("Invalid password");
    }
}
```

```

@PostLoad
@PostPersist
@PostUpdate
public void calculateAge() { ❷
    if (dateOfBirth == null) {
        age = null;
        return;
    }

    Calendar birth = new GregorianCalendar();
    birth.setTime(dateOfBirth);
    Calendar now = new GregorianCalendar();
    now.setTime(new Date());
    int adjust = 0;
    if (now.get(Calendar.DAY_OF_YEAR)
        - birth.get(Calendar.DAY_OF_YEAR) < 0) {
        adjust = -1;
    }
    age=now.get(Calendar.YEAR)-birth.get(Calendar.YEAR)+adjust;
}

public void matchPassword(String pwd) { ❸
    if (pwd == null || "".equals(pwd))
        throw new ValidationException("Invalid password");
    if (!pwd.equals(password))
        throw new ValidationException("Passwords don't match");
}

// constructeurs, accesseurs
// méthodes hashCode, equals et toString
}

```

◀ Lorsque l'entity est chargé (@PostLoad), ou après que ses données aient été insérées ou mises à jour, on calcule l'âge du client à partir de sa date de naissance.

◀ Cette méthode métier vérifie que le mot de passe passé en paramètre est valide, et qu'il est identique à celui stocké dans la base de données.

Adresse

L'entity Address est utilisé pour représenter l'adresse de domiciliation du client et l'adresse de livraison de la commande.

Code de l'entity Address

```

package com.yaps.petstore.entity;

@Entity
@Table(name = "t_address")
public class Address implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false)
    private String street1;
    private String street2;
    @Column(nullable = false, length = 100)
    private String city;
    private String state;
    @Column(name = "zip_code", nullable = false, length = 10)

```

◀ Entity Address rendant persistantes ses données dans la table t_address.

◀ Une adresse est composée d'un identifiant unique, de deux attributs pour stocker la rue, d'une ville, d'un état, d'un code postal et d'un pays.

Avant d'insérer ou de mettre à jour les données en base, cette méthode est appelée. Elle permet de valider les attributs de l'entity et renvoie une exception en cas d'incohérence.

```
private String zipcode;  
@Column(nullable = false, length = 50)  
private String country;  
  
@PrePersist  
@PreUpdate  
private void validateData() {  
    if (street1 == null || "".equals(street1))  
        throw new ValidationException("Invalid street");  
    if (city == null || "".equals(city))  
        throw new ValidationException("Invalid city");  
    if (zipcode == null || "".equals(zipcode))  
        throw new ValidationException("Invalid zip code");  
    if (country == null || "".equals(country))  
        throw new ValidationException("Invalid country");  
}  
  
// constructeurs, accesseurs  
// méthodes hashCode, equals et toString  
}
```

UML Stéréotypes

En UML, les stéréotypes permettent de créer de nouveaux éléments de modélisation. Ils sont représentés soit par un graphique (comme l'acteur dans les cas d'utilisation), soit entre guillemets. Dans nos diagrammes de classes, les entities sont stéréotypés <<entity>>.

Le bon de commande

Lorsque le client achète des animaux de compagnie, un bon de commande est automatiquement créé par le système. Un bon de commande (Order) est constitué de lignes de commandes (OrderLine). Il fait référence au client (Customer) qui a passé la commande. Il est payable par une carte bancaire (CreditCard).

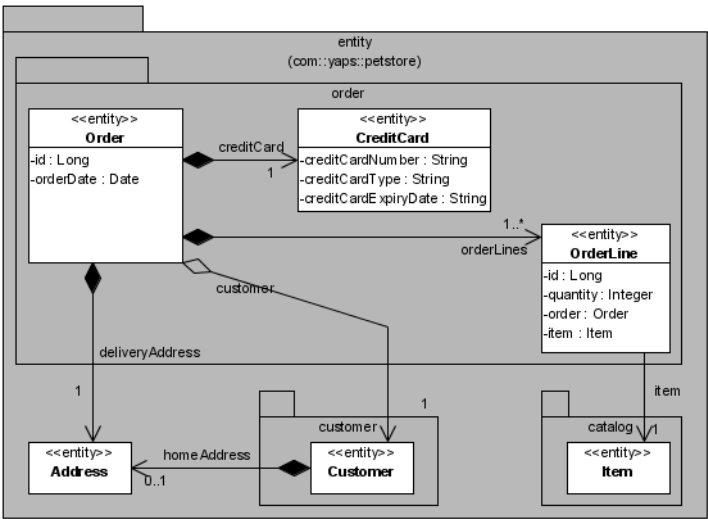


Figure 4-10
Diagramme de classes du bon de commande

Le diagramme de classes précédent nous montre les attributs du bon de commande (identifiant et date de création), des lignes de commandes

(quantité achetée et référence vers l'article acheté) et la carte de crédit (numéro de la carte, date de fin de validité et type).

Notez que ces entités ont des liens avec des paquetages externes (comme l'article qui se trouve dans le paquetage `catalog` par exemple). La classe `Address`, quant à elle, se trouve à la racine du paquetage `com.yaps.petstore.entity` puisqu'elle est commune au bon de commande et au client.

Bon de commande

Le bon de commande est relié à différents entités : au client ❶ qui a passé la commande, à l'adresse de livraison ❷ et à ses lignes de commandes ❸ (lien unidirectionnel). Quant à la carte de crédit, elle est englobée grâce à l'utilisation de l'annotation `@Embedded` ❹. Lors de l'insertion du bon de commande en base, on affecte la date de création à la date courante grâce à une méthode annotée par `@PrePersist` ❺. La méthode métier `getTotal` ❻ calcule le montant total du bon de commande, c'est-à-dire qu'elle cumule le montant de chaque ligne de commande.

Code de l'entity Order

```
package com.yaps.petstore.entity.order;

@Entity
@Table(name = "t_order")
public class Order implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "order_date", updatable = false)
    @Temporal(TemporalType.DATE)
    private Date orderDate;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "customer_fk", nullable = false)
    private Customer customer; ❶

    @OneToOne(fetch = FetchType.EAGER, cascade =
CascadeType.ALL)
    @JoinColumn(name = "address_fk", nullable = false)
    private Address deliveryAddress; ❷

    @Embedded
    private CreditCard creditCard = new CreditCard(); ❹

    @OneToMany(cascade = CascadeType.ALL,
        fetch = FetchType.EAGER)
    @JoinTable(name = "t_order_order_line",
        joinColumns = {@JoinColumn(name = "order_fk")},
        inverseJoinColumns={@JoinColumn(name="order_line_fk")})
    private List<OrderLine> orderLines; ❸
```

❶ Entity Order rendant persistantes ses données dans la table `t_order`.

❷ Le bon de commande possède un identifiant unique et une date de création.

❸ Lien unidirectionnel avec le client.

❹ Lien unidirectionnel avec l'adresse de livraison.

❺ Les données de la carte de crédit sont englobées dans la même table que le bon de commande.

❻ Le lien unidirectionnel avec les lignes de commandes est effectué à l'aide de la table de jointure `t_order_order_line`. On redéfinit les colonnes de clés étrangères grâce aux annotations `@JoinColumn`.

› Lors de l'insertion en base, la date de création du bon de commande est initialisée avec la date courante.

› Cette méthode métier calcule le montant total du bon de commande. Pour cela, elle itère la liste des lignes de commandes et cumule les sous-totaux (en appelant la méthode `getSubTotal`).

```
@PrePersist
private void setDefaultData() { ❸
    orderDate = new Date();
}

public Float getTotal() { ❹
    if (orderLines == null || orderLines.isEmpty())
        return 0f;

    Float total = 0f;
    for (OrderLine orderLine : orderLines) {
        total += (orderLine.getSubTotal());
    }

    return total;
}

// constructeurs, accesseurs
// méthodes hashCode, equals et toString
}
```

Ligne de commande

Un bon de commande est constitué d'une ou plusieurs lignes de commande. Chacune d'elles nous informe de la quantité d'articles achetés. Ainsi, une ligne de commande fait référence à un article ❶ par un lien unidirectionnel.

Code de l'entity OrderLine

```
package com.yaps.petstore.entity.order;

@Entity
@Table(name = "t_order_line")
public class OrderLine implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(nullable = false)
    private Integer quantity;

    @OneToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "item_fk", nullable = false)
    private Item item; ❶

    @PrePersist
    @PreUpdate
    private void validateData() {
        if (quantity == null || quantity < 0)
            throw new ValidationException("Invalid quantity");
    }

    public Float getSubTotal() {
        return item.getUnitCost() * quantity;
    }
}
```

› Entity `OrderLine` rendant persistant ses données dans la table `t_order_line`.

› Une ligne a un identifiant unique et la quantité d'articles achetés par le client est indiquée.

› L'article référencé par la ligne de commande est chargé de manière automatique.

› Avant d'insérer ou de mettre à jour les données en base, cette méthode est appelée. Elle permet de valider les attributs de l'entity et renvoie une exception en cas d'incohérence.

› Méthode métier calculant le sous-total d'une ligne de commande, c'est-à-dire le prix de l'article multiplié par la quantité achetée.

```
// constructeurs, accesseurs
// méthodes hashCode, equals et toString
}
```

Carte de crédit

L'entity `CreditCard` utilise l'annotation `@Embeddable` ❶ pour pouvoir être englobé par le bon de commande.

Code de l'entity `CreditCard`

```
package com.yaps.petstore.entity.order;

@Embeddable ❶
public class CreditCard implements Serializable {

    @Column(name = "credit_card_number", length = 30)
    private String creditCardNumber;
    @Column(name = "credit_card_type")
    private String creditCardType;
    @Column(name = "credit_card_expiry_date", length = 5)
    private String creditCardExpDate;

    @PrePersist
    @PreUpdate
    private void validateData() {
        if (creditCardNumber==null || "".equals(creditCardNumber))
            throw new ValidationException("Invalid number");
        if (creditCardType == null || "".equals(creditCardType))
            throw new ValidationException("Invalid type");
        if (creditCardExpDate==null|| "".equals(creditCardExpDate))
            throw new ValidationException("Invalid expiry date");
    }

    // constructeurs, accesseurs
    // méthodes hashCode, equals et toString
}
```

❖ Les données de la carte de crédit sont englobées par l'entity `Order`. Les données sont donc stockées dans la table `t_order`.

❖ La carte de crédit n'est pas une entity, elle n'a donc pas d'identifiant unique. Elle est constituée d'un numéro, d'un type (Visa, Master Card, etc.) et d'une date d'expiration au format mois/année (MM/AA).

❖ Avant d'insérer ou de mettre à jour les données en base, cette méthode est appelée. Elle permet de valider les attributs de l'entity et renvoie une exception en cas d'incohérence.

Paquetages des entities

Les classes de l'application YAPS Pet Store sont développées dans le paquetage `com.yaps.petstore`. La règle de nommage est la suivante : les paquetages des applications commerciales commencent par `com`, suivi du nom de l'entreprise (`yaps`) et du nom du projet (`petstore`). Les objets persistants que nous venons de voir se trouvent tous dans le paquetage `com.yaps.petstore.entity`.

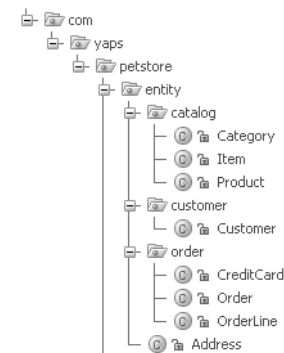


Figure 4–11 Paquetages et entities

Schéma de la base de données

Tous les entités de l'application YAPS Pet Store que nous venons de voir sont rendus persistants dans des tables. Leurs attributs et leurs relations nous donnent le schéma de base de données suivant :

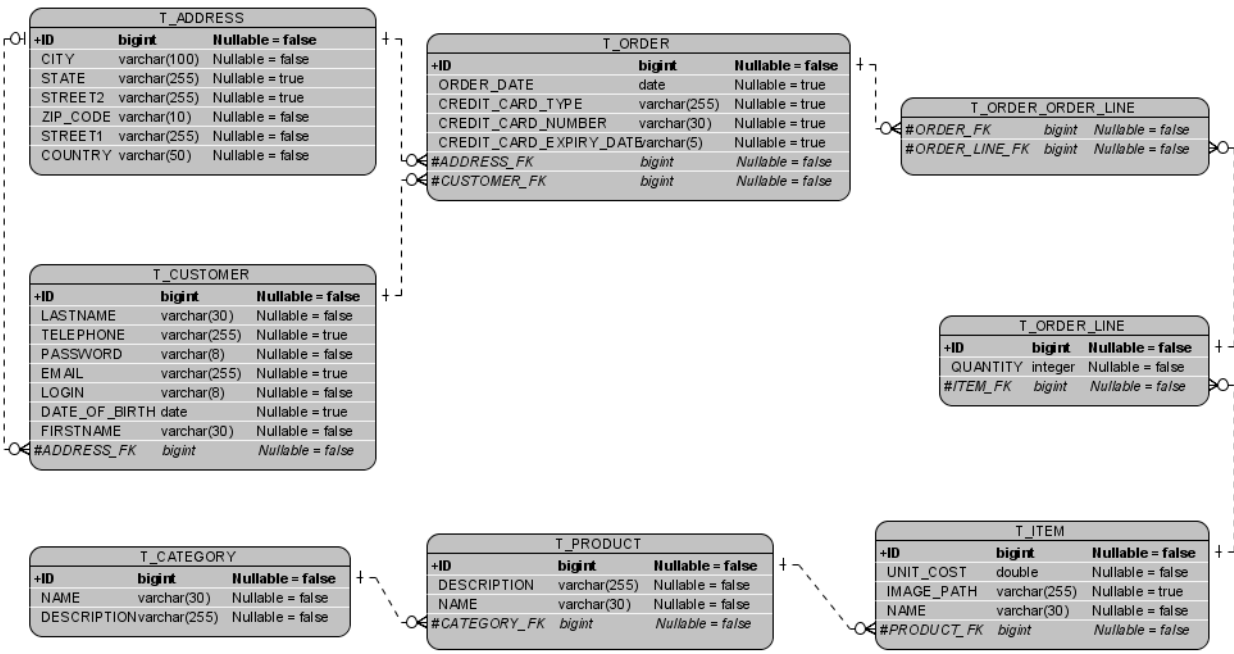
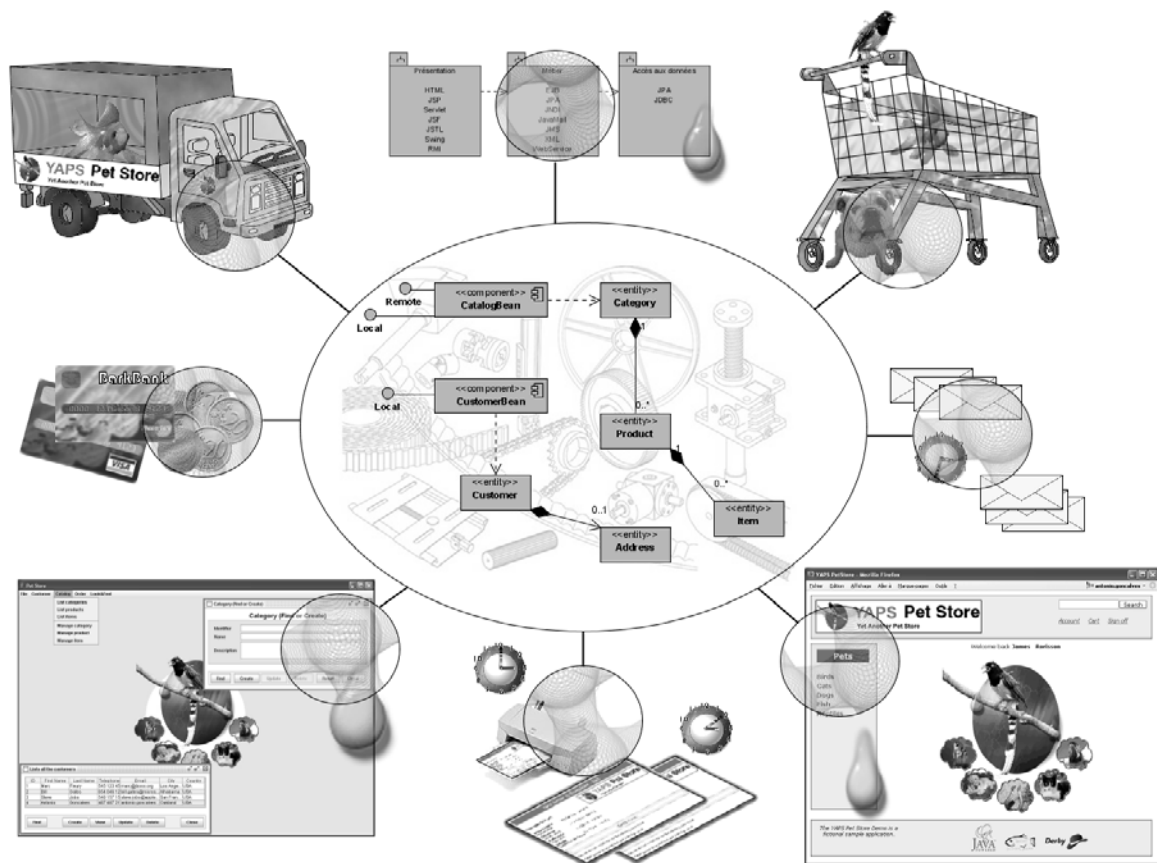


Figure 4-12 Schéma de la base de données

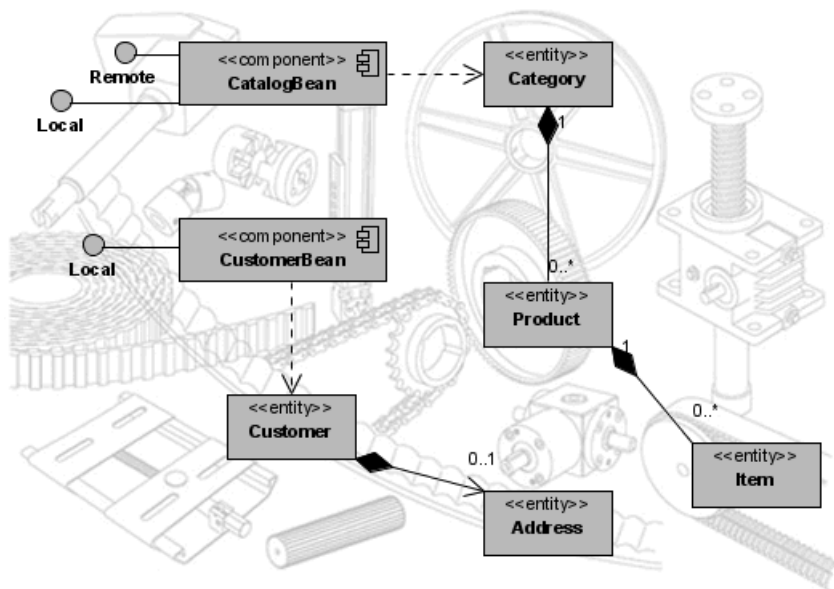
Notez la présence de la table de jointure `t_order_order_line` qui permet de stocker les clés étrangères du bon de commande et de ses lignes.

En résumé

Dans ce chapitre, nous avons développé les objets persistants de l'application YAPS Pet Store. Pour cela, nous avons utilisé JPA qui permet de simplifier le mécanisme de mapping objet-relationnel. Cette spécification présente l'avantage d'avoir réconcilié les détracteurs des entity beans 2.x avec Java EE sans décevoir les utilisateurs de frameworks Open Source dont elle s'est inspirée. Dans le chapitre suivant, nous découvrirons comment manipuler ces objets à partir d'une couche de traitements de stateless session beans.



chapitre 5



Traitements métier

Dans le chapitre précédent, nous avons implémenté les objets persistants de l'application sous forme d'entités. Nous allons maintenant développer la couche de traitements qui les manipulera, en utilisant les EJB session sans état (stateless). Cette couche se charge de gérer les exceptions, de démarquer les transactions, et utilise l'entity manager de JPA ainsi que son langage de requêtes JPQL pour accéder aux données.

SOMMAIRE

- ▶ Couche de traitements métier
- ▶ Gestion du catalogue et des clients
- ▶ Stateless session bean
- ▶ Cycle de vie des stateless beans
- ▶ Interfaces locale et distante
- ▶ Entity manager
- ▶ Langage de requêtes JPQL
- ▶ Gestion des exceptions
- ▶ Démarcation des transactions

MOTS-CLÉS

- ▶ Stateless bean
- ▶ Entity manager
- ▶ JPQL
- ▶ Contexte de persistance
- ▶ CRUD
- ▶ Logging

ARCHITECTURE Séparation des responsabilités

La séparation des responsabilités, ou *Separation of Concerns*, est le processus visant à découper un programme en éléments distincts dont les fonctionnalités se recouvrent le moins possible. Dans notre cas, les entités s'occupent de la persistance, les stateless des traitements et l'interface graphique de l'affichage. Ceci garantit le principe de séparation des données, des traitements et des interfaces pour faciliter la maintenance et la réutilisabilité de tout ou partie d'un logiciel.

RAPPEL CRUD

Opérations de base pour la persistance. Le terme CRUD signifie en anglais : *Create*, *Retrieve* (ou *Read*), *Update*, *Delete*, c'est-à-dire la création, la lecture, la mise à jour et la suppression de données.

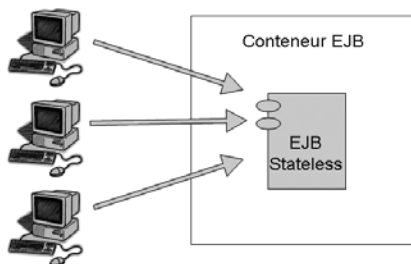


Figure 5-1

Plusieurs clients pour un même stateless bean

Les entités étudiés au chapitre précédent offrent un modèle de persistance objet. Ils encapsulent les données et leur mapping relationnel grâce aux annotations JPA, tout comme ils décrivent des méthodes métier qui leur sont propres. En revanche, les entités ne sont pas faits pour représenter les tâches complexes qui nécessiteraient une interaction avec d'autres objets persistants. Ce n'est pas la couche appropriée pour ce type de traitements. De la même manière, l'interface utilisateur ne doit pas comporter de logique métier (surtout lorsqu'on multiplie le nombre d'interfaces, Web et Swing, ce qui nous pousserait à dupliquer du code). Pour décorréler les objets persistants de la couche de présentation, nous utilisons une couche de traitements métier, implémentée sous forme de stateless beans.

Cette couche de traitements va agir comme un chef d'orchestre. Elle effectue des opérations de persistance sur les entités (CRUD), enchaîne les appels aux entités et rajoute de la logique métier.

Stateless session bean

Un stateless session bean, ou composant sans état, est un objet particulier qui réside dans un conteneur (on parle alors de conteneur d'EJB). Contrairement aux entités, les EJB stateless ne persistent pas de données, ils servent à exécuter des traitements. Comme son nom l'indique, le stateless session bean ne possède pas d'état. En effet, il sert à l'exécution d'un traitement et retourne un résultat sans avoir connaissance des appels précédents ou futurs. Par exemple, une application cliente contacte un stateless bean et lui transmet des paramètres. Le stateless bean sélectionne des entités correspondant aux paramètres, et retransmet un résultat au client. Lorsque ce traitement s'achève, le stateless bean ne conserve aucun souvenir de cette interaction.

Parce qu'il ne possède pas d'état, un stateless bean peut être utilisé par différents clients. Ainsi, le même EJB pourra retourner un résultat à un client puis, immédiatement après, exécuter le même traitement pour un autre client. Ceci implique qu'un appel de méthode doit passer tous les paramètres nécessaires à sa bonne exécution. Il est donc impossible d'appeler plusieurs méthodes pour construire un traitement. Un stateless bean n'est pas dédié à un seul client et ne conserve pas d'état (figure 5-1).

Le client d'un EJB peut être un programme de toute forme : une application avec ou sans interface graphique, une servlet, une JSP ou un autre EJB. Comme nous le verrons dans les prochains chapitres, les appels aux EJB, dans l'application YAPS Pet Store, seront effectués par le client graphique Swing et le client web (au travers de JSF). Cette couche de

traitements, inspirée du design pattern Façade, agit donc comme un point d'entrée unique pour les différents clients.

Le design pattern Façade

Le design pattern Façade du GoF (*Gang of Four*) permet de fournir un point d'entrée simple à un système complexe. Il introduit une interface pour découpler les relations entre deux systèmes : dans notre cas, les interfaces graphiques et les objets persistants. Grâce à ce design pattern, le code qui gère les appels aux différents objets est regroupé à un seul endroit et peut être réutilisé par différentes interfaces utilisateurs. Le succès de ce design pattern a son pendant pour les EJB où la problématique réside dans son insertion dans un environnement

distribué : il s'agit du pattern SessionFacade. Comme les EJB s'exécutent dans un conteneur, les appels sont donc réalisés à distance. C'est pourquoi les objets doivent être sérialisés puis transmis à travers un réseau. Le design pattern SessionFacade offre un point d'entrée unique et permet donc de limiter les appels distants. Dans notre architecture, les stateless session beans font office de SessionFacade et s'occupent de faire le lien entre les interfaces utilisateurs et les objets persistants.

Exemple de stateless bean

Pour développer un stateless bean, il faut une classe qui contienne le code métier et, au minimum, une interface permettant les appels. Dans l'exemple ci-après, l'interface distante `CustomerRemote` définit une méthode pour créer un nouveau client (`createCustomer`). La classe `CustomerBean`, quant à elle, implémente cette interface en ajoutant du code métier pour manipuler les entités `Customer` et `Address`.

Interface distante

```
@Remote ❶
public interface CustomerRemote {
    Customer createCustomer(Customer customer,
                           Address homeAddress) ;
}
```

Classe du stateless bean

```
@Stateless ❷
public class CustomerBean implements CustomerRemote ❸ {

    @PersistenceContext
    private EntityManager em; ❹

    public Customer createCustomer(Customer customer,
                                   Address homeAddress) {
        customer.setHomeAddress(homeAddress);
        em.persist(customer); ❺
        return customer; ❻
    }
    // Autres méthodes métier
}
```

REMARQUE EJB ou stateless bean ?

Le sigle EJB signifie *Enterprise Java Bean* et correspond aux différents composants de la plateforme Java EE. Ce terme générique englobe les stateless beans qui peuvent être appelés EJB stateless, stateless beans ou stateless session beans.

Cet exemple de code représente le composant `CustomerBean` qui gère la création d'un nouveau client. Tout d'abord, notez la présence de l'interface `CustomerRemote`. Celle-ci est annotée par `@javax.ejb.Remote` ❶ qui signifie que toutes les méthodes qu'elle déclare (dans notre exemple `createCustomer`), peuvent être appelées de manière distante. La classe `CustomerBean` implémente cette interface ❸ et est donc contrainte d'ajouter du code métier à la méthode `createCustomer`. Il est intéressant de constater l'utilisation de l'annotation `@javax.ejb.Stateless` ❷. Sans cette annotation, la classe `CustomerBean` serait considérée comme toute autre classe Java. Grâce à elle, le conteneur d'EJB sait qu'il manipule un stateless bean.

Attardons-nous maintenant sur l'implémentation de la méthode `createCustomer`. Son rôle est de créer un nouveau client dans la base de données. Pour cela, cette méthode prend en paramètres les deux entités `Customer` et `Address` et utilise un `EntityManager` ❹ pour les persister ❺ (le rôle de l'`EntityManager` sera détaillé dans les prochains paragraphes). Une fois l'opération effectuée avec succès, l'entité `Customer` est retourné ❻ en résultat de la méthode.

Comment développer un stateless bean

Dans le chapitre précédent, intitulé *Objets persistants*, nous avons constaté que le développement d'un entity était relativement simple et surtout, identique à n'importe quelle classe Java (hormis l'utilisation de quelques annotations). Pour un stateless bean, c'est tout aussi facile puisqu'il suffit de développer une classe ainsi qu'une ou deux interfaces.

Les interfaces

Un stateless bean peut avoir une interface distante et/ou locale. L'interface distante (`@javax.ejb.Remote`) permet aux clients distants d'invoquer des méthodes de l'EJB. Les paramètres des méthodes sont ainsi copiés, sérialisés, puis transmis à l'EJB. On appelle ce mécanisme l'appel par valeur (*call-by-value*).

L'interface locale (`@javax.ejb.Local`) est utilisée par les objets résidants dans la même JVM que l'EJB. Les paramètres des méthodes ne sont pas recopiés mais passés par référence (*call-by-reference*).

JAVA Passage par valeur et référence

Le passage de paramètres en local dans Java se fait presque toujours par référence. Seul un pointeur (référence) vers un objet est passé à la méthode. Par contre, les données de type primitif (`byte`, `short`, `char`...) sont passées par valeur, c'est-à-dire qu'une copie de la donnée est mise à disposition de la méthode et non l'originale.

La plupart de nos stateless session beans utilisent les deux interfaces. En effet, l'application YAPS Pet Store est constituée d'une interface graphique de type client lourd (Swing), utilisée par les employés et qui s'appuiera sur les interfaces @Remote, ainsi qu'une application web, hébergée sur le même serveur que les EJB, qui, quant à elle, utilisera les interfaces @Local. Les deux interfaces peuvent exposer des méthodes différentes.

Interface distante

L'interface distante définit les méthodes de l'EJB accessibles en dehors du conteneur (application Swing utilisée par l'employé). Cette interface est annotée par @javax.ejb.Remote.

Reprenons l'exemple du CustomerBean. Ci-après le code de l'interface CustomerRemote :

Interface distante

```
@Remote
public interface CustomerRemote {
    Customer createCustomer(Customer customer,
                           Address homeAddress) ;
}
```

Comme vous pouvez le constater, cette interface ressemble de très près à n'importe quelle interface Java. La seule différence est la présence de l'annotation @javax.ejb.Remote qui informe le conteneur que cette interface peut être appelée de manière distante.

Les paramètres des méthodes distantes doivent être sérialisables pour être véhiculés à travers le réseau. Rappelez-vous que les entités que nous avons développées implémentent l'interface Serializable.

Code de l'annotation @javax.ejb.Remote

```
package javax.ejb;

@Target({TYPE}) @Retention(RUNTIME)
public @interface Remote {
    Class[] value() default {};
}
```

EJB Interface distante dans un cluster

On utilise un cluster (groupe d'ordinateurs vu comme une seule machine) pour des raisons de performance, de répartition de charge, ou de tolérance aux pannes. Dans ce cas, il est nécessaire d'utiliser des interfaces distantes pour que les EJB puissent communiquer entre eux dans le cluster.

EJB RemoteException

Contrairement aux EJB 2.1, les méthodes n'ont pas besoin de lancer l'exception java.rmi.RemoteException. Il est toujours possible de le faire, mais ce n'est plus obligatoire depuis la version 3.0 des EJB.

EJB Les stateless beans 2.x

Pour vous faire une idée des modifications apportées à la spécification EJB, retrouvez en annexe le code source d'un stateless bean 2.1.

◀ Cette annotation s'applique à une classe.

◀ Spécifie la liste des interfaces distantes sous forme de tableau de classes. Cet attribut est utilisé si la classe du bean implémente plus d'une interface distante.

APPROFONDIR Que signifie locale ?

Bien que nous n'ayons pas encore vu le déploiement de l'application, la notion d'interface locale est fortement liée à ce processus. L'application YAPS Pet Store va être déployée dans un fichier ear (Enterprise Archive) qui contient tous les composants de l'application (EJB, application web, entités...). L'interface locale d'un EJB est visible et utilisable à l'intérieur de ce fichier ear. Si on déploie deux applications sur le même serveur (A.ear et B.ear), l'interface locale d'un EJB dans A ne pourra pas être vue par un EJB dans B. La visibilité n'est donc pas directement liée au serveur d'applications mais à l'Enterprise Archive (ear).

DTO : design pattern ou anti-pattern

Aussi connu sous le nom de Value Object (VO), le design pattern Data Transfert Object (DTO) permet de transporter des données du client au serveur et inversement en réduisant les appels réseaux. Ce design pattern est communément utilisé dans les architectures EJB 2.x. En effet, les entity beans étant représentés par des composants riches (*heavyweight*), qui ne sont pas sérialisables – rappelez-vous les interfaces `javax.ejb.EJBLocalHome` ou `javax.ejb.EJBLocalObject` –, il est impossible de les manipuler en dehors du conteneur. Les objets DTO, composés uniquement d'attributs et d'accesseurs, collectent des données de plusieurs entity beans pour en obtenir une vision uniforme et permettre leur manipulation par des stateless beans. Dépourvus d'accès à des ressources externes, de logique métier, ou de persistance, ces DTO sont chargés à partir des attributs des entity beans, puis transmis au client pour enfin revenir au serveur. Ainsi, avec un seul appel distant, on obtient un DTO qui contient toutes les informations souhaitées.

Ce design pattern peut maintenant être vu comme un anti-pattern dans l'architecture EJB 3 s'il est utilisé constamment. Comme nous l'avons vu, un entity est maintenant une simple classe Java et peut être facilement manipulé comme tel. De plus, comme nous allons le voir, les entités peuvent être détachés de leur gestionnaire de persistance pour traverser les différentes couches de l'application et, être vus comme de simples Pojo.

Bien entendu, le design pattern DTO peut toujours être utilisé dans certains cas, comme celui des services web, qui doivent fournir une interface stable (évoluant moins rapidement que celle des EJB). Il n'est plus une pièce indispensable de l'architecture, mais tout simplement une solution supplémentaire utilisable dans une situation particulière.

- <http://java.sun.com/blueprints/corej2eepatterns/Patterns/TransferObject.html>
- <http://c2.com/cgi/wiki?DataTransferObject>

Interface locale

L'interface locale définit les méthodes accessibles à l'intérieur du serveur d'applications, c'est-à-dire par d'autres EJB ou applications web hébergées sur le même serveur. Cette interface permet les appels locaux sans rajouter de surcoût lié à la sérialisation des paramètres qui devient alors inutile. Cette interface est identifiée par l'annotation `@javax.ejb.Local`. Ci-après, l'interface `CustomerLocal` définissant la même méthode pour créer un client.

Interface locale

```
@Local
public interface CustomerLocal {
    Customer createCustomer(Customer customer,
                           Address homeAddress) ;
}
```

Comme vous pouvez le constater, le code est identique à celui de l'interface distante. La seule différence réside dans l'utilisation de l'annotation `@Local` au lieu de `@Remote`.

Code de l'annotation `@javax.ejb.Local`

```
package javax.ejb;

@Target({TYPE}) @Retention(RUNTIME)
public @interface Local {
    Class[] value() default {};
}
```

◀ Cette annotation s'applique à une classe.

◀ Spécifie la liste des interfaces locales sous forme de tableau de classes. Cet attribut est utilisé si la classe du bean implémente plus d'une interface locale.

La classe de l'EJB

Cette classe contient la logique métier du composant et doit avoir au moins une interface. Si la classe implémente les deux interfaces, cela signifie que le même EJB peut être appelé de manière locale et distante. Dans les deux cas, le client de l'EJB n'utilise pas directement cette classe, mais doit passer par une des interfaces.

La classe d'implémentation du bean

```
@Stateless
public class CustomerBean implements CustomerRemote,
                                   CustomerLocal {

    @PersistenceContext
    private EntityManager em;

    public Customer createCustomer(Customer customer,
                                   Address homeAddress) {

        customer.setHomeAddress(homeAddress);
        em.persist(customer);
        return customer;
    }
    // Autres méthodes métier
}
```

SYNTAXE Classe EJB

La classe de l'EJB doit suivre les règles de développement suivantes :

- elle doit être annotée par `@Stateless` ;
- elle doit être publique ;
- elle ne doit pas être finale ;
- elle ne doit pas être abstraite ;
- elle ne doit pas définir la méthode `finalize` ;
- elle doit avoir un constructeur par défaut ;
- elle doit implémenter les méthodes de ses interfaces.

Comme vous le voyez ci-dessus, pour distinguer un Pojo d'un EJB stateless, il faut utiliser l'annotation `@javax.ejb.Stateless`. Dans cet exemple, la classe implémente les deux interfaces et doit donc définir la méthode `createCustomer`.

Cette annotation s'applique à une classe.	▶
Nom de l'EJB. Par défaut, le nom est celui de la classe.	▶
Cet attribut représente le nom donné à l'EJB à l'intérieur du conteneur. Il est spécifique à chaque serveur d'applications et peut donc ne pas être portable.	▶
Description du stateless session bean.	▶

EJB Un nom JNDI

Nous développerons l'utilisation et le fonctionnement de JNDI au chapitre suivant, *Exécution de l'application*. Je tiens juste à vous faire remarquer que pour modifier le nom JNDI par défaut de l'EJB dans GlassFish, il faut utiliser l'attribut `mappedName` de l'annotation `@Stateless`. Le code ci-dessous attribue le nom `ejb/stateless/Custom` au `CustomerBean`.

```
@Stateless(mappedName =  
    "ejb/stateless/Custom")  
public class CustomerBean {}
```

Code de l'annotation @javax.ejb.Stateless

```
package javax.ejb;  
  
@Target(value = {TYPE}) @Retention(value = RUNTIME)  
public @interface Stateless {  
  
    String name() default "";  
  
    String mappedName() default "";  
  
    String description() default "";  
}
```

La classe de notre exemple précédent implémente les deux interfaces. En EJB 3.0, la classe a le choix entre implémenter les interfaces, ou les définir grâce aux annotations `@Remote` et `@Local`.

```
@Remote(value = CustomerRemote.class)  
@Local(value = CustomerLocal.class)  
@Stateless  
public class CustomerBean {  
    (...)  
}
```

L'avantage d'implémenter les interfaces est que le compilateur peut déceler les méthodes qui seraient définies dans les interfaces mais pas dans la classe d'implémentation. Cela réduit donc les erreurs de déploiement.

Entity manager

Comme énoncé en introduction de ce chapitre, un des rôles de la couche de traitements est de manipuler les entités. Ceux-ci sont de simples classes Java que l'on peut instancier à l'aide de l'opérateur `new`. En revanche, quand on veut les persister en base de données, il faut utiliser un `entity manager`.

Pour ceux d'entre vous qui ont déjà manipulé les `entity beans 2.x`, vous vous souvenez peut-être qu'ils utilisent une `Home Interface`. Cette interface permet de créer, mettre à jour et supprimer les entités de la base de données. En EJB 3, les entités sont de simples `Pojo` et n'implémentent aucune interface. Il leur faut donc utiliser les services de la classe `javax.persistence.EntityManager`.

Dans JPA, l'entity manager est le service centralisant toutes les actions de persistance. Les entités ne deviennent persistantes que lorsqu'on le précise explicitement dans le code au travers de l'entity manager. Celui-ci fournit une API pour créer, rechercher, mettre à jour, supprimer et synchroniser des objets avec la base de données. En fait, l'entity manager peut être vu comme un DAO (*Data Access Object*) générique : il permet d'effectuer les méthodes CRUD sur n'importe quel entity.

Pour instancier un entity en mémoire, il faut utiliser le mot-clé `new`. Ensuite, pour que les données soient stockées en base, il faut utiliser la méthode `persist()` ③ de l'entity manager ②.

Utilisation de l'entity manager dans un stateless bean

```
@Stateless
public class CustomerBean implements CustomerRemote {
    @PersistenceContext (unitName = "petstorePU") ①

    private EntityManager em; ②

    public Customer createCustomer(Customer customer,
                                   Address homeAddress) {
        customer.setHomeAddress(homeAddress);

        em.persist(customer); ③
        return customer;
    }
}
```

Lorsqu'un entity est pris en compte par l'entity manager, on dit qu'il est attaché (ou managé). On peut alors effectuer des opérations de persistance. L'entity manager synchronise automatiquement l'état de l'entity avec la base de données. Lorsque l'entity se détache (il n'est plus pris en compte par l'entity manager), il devient un simple Pojo et peut ainsi être utilisé par les autres couches (par le client Swing, par exemple).

Contexte de persistance

L'entity manager est la pièce centrale servant à manipuler les entités. Grâce aux annotations JPA, il sait comment faire le mapping entre un entity et une table et, plus précisément, entre un attribut et une colonne. Par contre, il lui manque toujours une information : dans quelle base de données doit-il persister ces entités ?

Pour cela, il utilise un contexte de persistance ① qui le renseigne sur plusieurs informations : le type de la base de données (dans notre cas, on utilise Derby) et les paramètres de connexion à cette base de données (via l'utilisation d'une source de données).

APPROFONDIR JPA hors conteneur

La classe `EntityManager` peut être utilisée en dehors d'un conteneur dans une application Java SE. La seule différence est l'utilisation d'une classe `Factory` (`EntityManagerFactory`). Hormis cette classe, le code est identique en Java EE ou en Java SE.

► <http://www.oracle.com/technology/products/ias/toplink/jpa/howto/java-se-usage.html>

◀ Le contexte de persistance informe l'entity manager du type de base de données et des paramètres de connexion.

◀ Déclaration de l'entity manager.

◀ L'entity manager est utilisé pour persister l'entity `customer` dans la base de données.

RAPPEL Source de données

Une source de données, ou `Data Source`, représente une connexion physique à une base. Lorsque nous avons configuré le serveur GlassFish, nous avons créé la source de données `jdbc/petstoreDS` qui pointe vers la base `petstoreDB` dans Derby.

Remarquez l'annotation `@javax.persistence.PersistenceContext` ❶ apposée sur l'entity manager. Grâce à l'attribut `unitName = "petstorePU"`, l'entity manager sait faire le lien avec une unité de persistance qui se nomme `petstorePU`. Cette unité de persistance est définie dans le fichier `persistence.xml`. Comme nous le verrons plus loin, ce fichier doit être déployé dans le même jar que les entités.

Unité de persistance décrit dans le fichier `persistence.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>

  <persistence-unit name="petstorePU"> ❷
    <jta-data-source>jdbc/petstoreDS</jta-data-source> ❸

    <properties>
      <property name="toplink.target-database" value="Derby"/> ❹
    </properties>

  </persistence-unit>
</persistence>
```

L'attribut `name` ❷ de la balise `<persistence-unit>` est la référence utilisée dans le stateless bean ❶. Elle permet à l'EJB de connaître le type de la base de données ❹, c'est-à-dire Derby, et la manière de s'y connecter ❸ (via l'utilisation de la source de données `jdbc/petstoreDS` que nous avons créée dans GlassFish).

Manipuler les entités

L'interface `javax.persistence.EntityManager` fournit une API pour manipuler les entités. Voici un extrait ci-après présentant les principales méthodes que nous utiliserons dans la suite de ce chapitre.

Extrait de l'API de l'entity manager

```
package javax.persistence;
public interface EntityManager {

  void persist(Object object);
  <T> T merge(T t);
  void remove(Object object);
  <T> T find(Class<T> aClass, Object object);
  void flush();
  void clear();
  Query createQuery(String query);
}
```

Les méthodes `persist`, `merge`, `remove`, `find`, `flush` et `clear` permettent de manipuler un entity. `createQuery` est utilisée pour faire des requêtes sur des objets. Détaillons comment utiliser cette API.

Persister un entity

Persister un entity signifie que les valeurs des attributs sont stockés en base de données. On persiste des entités qui n'existent pas en base, sinon, on les met à jour. Pour ce faire, il faut créer une nouvelle instance de l'entité à l'aide de l'opérateur `new` ❶, affecter les valeurs grâce aux méthodes `set` ❷, lier un entity à un autre lorsqu'il y a des associations, et enfin, appeler la méthode `EntityManager.persist()` ❸.

Exemple simple de persistance d'un objet

```
Customer customer = new Customer(); ❶
customer.setId(1234);
customer.setFirstname("Paul"); ❷
customer.setLastname("Dupond");
em.persist(customer); ❸
```

Vous l'aurez compris, l'objet `Customer` étant un entity, il définit sa stratégie de mapping (grâce aux annotations que nous avons vues dans le précédent chapitre) et saura persister ses attributs dans la table adéquate.

Revenons à nos stateless beans. Dans notre architecture distribuée, ce sont les interfaces graphiques qui effectuent les `new` et les affectations de valeurs. Le rôle du stateless bean est donc de récupérer des objets déjà initialisés et de les persister.

Création d'un client dans un stateless bean

```
public Customer createCustomer(Customer customer, ❹
                                Address homeAddress) {

    customer.setHomeAddress(homeAddress); ❺
    em.persist(customer); ❻
    return customer;
}
```

Dans le code ci-dessus, la méthode permettant de créer un client possède deux entités en paramètres : un client et une adresse. Ces deux entités ont été instanciés par une application cliente (Swing, par exemple), qui a affecté les valeurs à l'aide des setters. L'appel à l'EJB est effectué en passant ces entités en paramètres ❹. Ensuite, le stateless bean a en charge de relier les beans entre eux ❺ et de les persister ❻ à l'aide de l'entity manager (la variable `em`). À ce moment là, l'objet `Customer` devient managé (attaché) et donc éligible pour être inséré en base.

PERSISTANCE Insertion en cascade

Si vous vous reportez au code de l'entity `Customer`, vous verrez qu'il utilise une relation en cascade avec l'adresse. `CascadeType.ALL` englobe le mode `CascadeType.PERSIST`. Ceci signifie que lorsqu'on insère un client, son adresse est aussi automatiquement insérée. Si ce n'avait pas été le cas, nous aurions dû persister l'adresse, puis le client comme ceci :

```
em.persist(address);
customer.setHomeAddress(address);
em.persist(customer);
```

Rechercher un entity par son identifiant

Pour rechercher un entity par son identifiant, on utilise la méthode `EntityManager.find()`. Cette méthode prend en paramètre la classe de l'entity ainsi que son identifiant et retourne un entity ❶. Si celui-ci n'est pas trouvé, la méthode `find` retourne la valeur `null`.

Exemple de recherche d'un objet après création

```
Customer customer = new Customer();
customer.setId(1234);
customer.setFirstname("Paul");
customer.setLastname("Dupond");
em.persist(customer);

customer = em.find(Customer.class, 1234) ❶
```

Pour rechercher un client, le stateless bean doit déclarer une méthode (`findCustomer` dans notre exemple) qui prend en paramètre un identifiant ❷ et retourne un entity ❸ (cette valeur de retour peut être égale à `null` si l'objet n'est pas trouvé). La recherche s'effectue via l'appel de la méthode `find` de l'entity manager ❹.

Recherche d'un client dans un stateless bean

```
public Customer ❸ findCustomer(Long customerId ❷ ) {

    Customer customer = em.find(Customer.class, customerId); ❹
    return customer;
}
```

Rattacher un entity

Si vous vous reportez au cycle de vie des entities que nous avons décrit dans le précédent chapitre, vous verrez qu'un entity peut être attaché (managé) ou détaché. Lorsqu'il est manipulé par une application graphique par exemple, il est détaché de sa persistance. De retour sur le serveur, il doit être rattaché à l'entity manager pour resynchroniser ses données avec la base. Pour ce faire, il suffit d'utiliser la méthode `EntityManager.merge()` qui prend en paramètre l'entity à rattacher.

Exemple simple de rattachement d'un objet

```
em.merge(customer)
```

Mettre à jour un entity

La mise à jour d'un entity est à la fois très simple à mettre en œuvre mais parfois difficile à comprendre car elle n'implique pas directement l'entity manager. Rien ne vaut un exemple. Le code ci-après recherche un entity ❶ et met à jour ses attributs en utilisant les méthodes set ❷.

Exemple de mise à jour d'un objet après recherche

```
Customer customer = em.find(Customer.class, 1234); ❶
customer.setFirstname("Antonio"); ❷
customer.setLastname("Goncalves");
```

La mise à jour est faite. Tout simplement. Il n'y a pas de mot-clé particulier pour mettre à jour les données d'un entity : après avoir effectué le find, le bean est toujours géré par l'entity manager. En appelant les méthodes set, qui mettent à jour les attributs du bean, l'entity manager sait qu'il doit synchroniser ces changements avec la base de données car le contexte de persistance est toujours actif.

Pour illustrer ce comportement, imaginez qu'après le find on réinitialise ❸ le contexte de persistance (EntityManager.clear()). L'entity n'étant plus géré, l'appel aux méthodes set mettra à jour les attributs de l'objet, mais ces mises à jour ne seront pas synchronisées dans la base de données.

Exemple de mise à jour qui échoue

```
Customer customer = em.find(Customer.class, 1234);
em.clear(); ❸
customer.setFirstname("Antonio");
customer.setLastname("Goncalves");
```

Revenons à notre stateless bean. Les entities sont instanciés et leurs attributs mis à jour par les interfaces graphiques. L'appel aux méthodes set s'effectue donc sur des entities détachés. Comment rendre effectifs ces changements ? Tout simplement en rattachant l'entity à l'aide de la méthode EntityManager.merge().

Dans le code suivant, la méthode pour mettre à jour un client possède deux paramètres : un entity Customer et un entity Address. Les attributs de ces deux entities ont été mis à jour par l'application cliente à l'aide des setters. L'appel à l'EJB est effectué en passant ces entities en paramètres ❹. Ensuite, le stateless bean a en charge de relier les beans entre eux ❺ et de les rattacher ❻ à l'entity manager qui synchronisera les changements en base de données.

PERSISTANCE

Avantages des outils de mapping

Pour manipuler les entities, nous n'avons écrit aucun ordre SQL. Dans l'exemple de la mise à jour, on voit bien qu'il est plus facile d'appeler une méthode set plutôt que d'écrire un ordre SQL update.

PERSISTANCE Synchronisation avec la base de données

Lorsque vous appelez les méthodes `persist()`, `merge()` ou `remove()`, l'entity manager peut, soit agir immédiatement sur la base de données, soit différer cette action à un moment qu'il juge opportun. Vous pouvez néanmoins forcer cette synchronisation en appelant la méthode `flush()`.

PERSISTANCE Les langages de requêtes

Les bases de données relationnelles utilisent le *Standard Query Language* (SQL).

► <http://www.w3schools.com/sql/default.asp>
La version 2.x des entity beans utilise l'EJBQL (*Enterprise Java Beans Query Language*).

► http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/EJBQL.html

Alors que JPA utilise le *Java Persistence Query Language* (JPQL).

► <http://java.sun.com/javaee/5/docs/tutorial/doc/bnbtg.html>

Mise à jour d'un client dans un stateless bean

```
public Customer updateCustomer(Customer customer, ④
                                Address homeAddress) {

    customer.setHomeAddress(homeAddress); ⑤
    em.merge(customer); ⑥
    return customer;
}
```

Supprimer un entity

Un entity peut être supprimé grâce à la méthode `EntityManager.remove()`. Cette dernière prend en paramètre l'entity, et entraîne la suppression des données en base. Une fois supprimé, l'entity se détache de l'entity manager et ne peut plus être manipulé par ce dernier. Le code suivant nous montre comment supprimer un objet après l'avoir trouvé.

Exemple de suppression d'un objet après recherche

```
customer = em.find(Customer.class, 1234)
em.remove(customer);
```

Revenons à notre architecture distribuée. L'entity à supprimer se trouve dans la couche de présentation. Le client appelle le stateless bean en passant l'entity en paramètre. La suppression ne peut alors s'effectuer qu'en deux étapes : rattacher l'objet à l'entity manager via la méthode `merge`, puis le supprimer en appelant la méthode `remove`. Ce mécanisme est illustré dans la méthode de l'EJB ci-dessous ①.

Suppression d'un client dans un stateless bean

```
public void deleteCustomer(Customer customer) {
    em.remove(em.merge(customer)); ①
}
```

Langage de requêtes

La possibilité d'effectuer des requêtes est inhérente aux bases de données relationnelles. Cela permet d'obtenir des informations répondant à certains critères, nécessaires au système. SQL, ou *Standard Query Language*, est le langage standard d'interrogation de base de données. Les résultats obtenus sont sous forme de table, c'est-à-dire de lignes et de colonnes.

JPA est une API de mapping objet-relationnel qui gère des objets et non des lignes et des colonnes. Pour conserver cette possibilité d'effectuer des requêtes sur les entities, JPA utilise son propre langage : JPQL.

JPQL

JPQL, ou Java Persistence Query Language, est un langage de requêtes déclaratif s'inspirant de la syntaxe de SQL. Sa particularité est de manipuler des objets dans sa syntaxe de requête et de retourner des objets en résultat. On manipule donc des objets dans une requête JPQL, puis le mécanisme de mapping transforme cette requête JPQL en langage compréhensible par une base de données relationnelle (en SQL).

L'avantage de JPQL est que le développeur n'a pas à connaître un nouveau langage. Comme nous le verrons dans les exemples à venir, JPQL est plus intuitif pour un développeur Java, car il utilise une approche objet. En effet, le développeur manipule son modèle objet, et non une structure de données, en utilisant la notation pointée (c'est-à-dire `maClasse.monAttribut`).

JPQL crée donc une abstraction par rapport à la base de données. Il est portable quel que soit le moteur utilisé via son interface Query. Si vous souhaitez utiliser les spécificités d'une base de données, JPQL vous permet de le faire en utilisant son interface NativeQuery.

PERSISTENCE NativeQuery

JPQL a une syntaxe très riche qui vous permet de manipuler les objets sous toute forme, et cela, de manière standard à toutes les bases de données. En revanche, si vous voulez utiliser une fonctionnalité propriétaire à une base de données, c'est-à-dire non portable, vous pouvez utiliser les NativeQuery. Elles permettent de prendre avantage de ces particularités tout en continuant à manipuler des entités.

Nous ne verrons pas de requêtes natives dans ce livre. Vous pouvez consulter le lien suivant pour plus d'informations.

► http://blogs.sun.com/JPQL01/entry/native_query_in_java_persistence

Effectuer des requêtes en JPQL

Les requêtes JPQL se font à l'aide de l'interface `javax.persistence.Query`. Cette interface est utilisée pour contrôler l'exécution d'une requête JPQL. L'entity manager fabrique un objet Query à partir d'un ordre JPQL, et le retourne pour qu'il soit ensuite manipulé par le programme.

Dans l'exemple de code suivant, l'entity manager (variable `em`) crée une Query à l'aide d'une chaîne de caractères ❶. Cette Query est ensuite utilisée pour passer des paramètres à la requête ❷ avant d'être exécutée ❸ et de retourner un entity.

```
Query query = em.createQuery("SELECT c FROM Customer c ❶
                             WHERE c.login=:param");
query.setParameter("param", login); ❷
Customer customer = (Customer) query.getSingleResult(); ❸
```

Regardons de plus près cette chaîne de caractères. Celle-ci est constituée de trois mots-clés : `SELECT`, `FROM` et `WHERE`. Ils permettent de sélectionner un objet `c` de type `Customer`, dont l'attribut `login` (`c.login` ou `c.getLogin()`) est égal à un paramètre (`:param` – notez le préfixe `:`). Ce paramètre, que l'on nomme arbitrairement `param`, est affecté dans la ligne suivante ❷. La requête, ainsi constituée, est exécutée via la méthode `getSingleResult` qui retourne un `Object`, lui-même « casté » en `Customer`. Nous obtenons ainsi un entity `Customer` dont le `login` est égal à un paramètre passé à cette méthode.

La requête retourne une liste d'objets.

La requête retourne un objet unique.

Il existe plusieurs méthodes pour passer des paramètres à la requête.

Extrait de l'interface javax.persistence.Query

```
package javax.persistence;
public interface Query {
    List getResultList();
    Object getSingleResult();
    Query setParameter(String name, Object value);
    Query setParameter(String name, Date date,
        TemporalType temporalType);
    Query setParameter(String name, Calendar calendar,
        TemporalType temporalType);
    Query setParameter(int i, Object value);
    Query setParameter(int i, Date date,
        TemporalType temporalType);
    Query setParameter(int i, Calendar calendar,
        TemporalType temporalType);
}
```

JPQL est un langage extrêmement riche. Il permet de faire toutes sortes de requêtes sur un modèle objet aussi compliqué soit-il (association entre classes, héritage, classe abstraite, interface...). On peut ainsi faire des jointures entre entités (LEFT JOIN, JOIN FETCH) ou effectuer des sous-requêtes. Comme en SQL, il existe des opérateurs pour filtrer les résultats (IN, NOT IN, EXIST, LIKE, IS NULL, IS NOT NULL), pour ne pas ramener de doublons (DISTINCT) ou pour contrôler la taille des collections (IS EMPTY, IS NOT EMPTY, CONTAINS). JPQL vient aussi avec toute une batterie de fonctions pour les chaînes de caractères (LOWER, UPPER, TRIM, CONCAT, LENGTH, SUBSTRING), les numériques (ABS, SQRT, MOD), ou pour les ensembles (COUNT, MIN, MAX, SUM). Comme en SQL, on peut trier les résultats (ORDER BY) ou les regrouper (GROUP BY).

JPQL mériterait un chapitre à lui tout seul, mais ce n'est pas l'objet de ce livre. Nous allons donc uniquement analyser les requêtes utilisées dans notre application.

L'application YAPS Pet Store a souvent besoin d'afficher la totalité des objets se trouvant dans la base de données (toutes les catégories ou tous les clients). En JPQL, il suffit d'écrire une requête sans restreindre le résultat (sans clause WHERE). Par exemple, pour avoir la liste de tous les clients :

```
Query query = em.createQuery("SELECT c FROM Customer c");
List<Customer> customers = query.getResultList();
```

ou alors tous les clients triés par leur nom de famille.

```
Query query = em.createQuery("SELECT c FROM Customer c
    ORDER BY c.lastname");
List<Customer> customers = query.getResultList();
```

Approfondir JPQL

JPQL est décrit dans la spécification de persistance EJB 3 et discuté sur le site de Sun.

- ▶ <http://jcp.org/en/jsr/detail?id=220>
- ▶ <http://wiki.java.net/bin/view/Javapedia/JPQL>

EXEMPLES **Ordres JPQL**

```
Voici quelques exemples d'ordres JPQL :
select o1 from Order o1, Order o2 where o1.quantity > o2.quantity
select c from Customer c where p.firstname = 'Paul'
select a from Address a where a.country in ('USA', 'PT', 'FR')
select o from Order o where o.orderLines is empty
```

La requête la plus compliquée de l'application concerne la recherche d'animaux domestiques. Si l'on se reporte au cas d'utilisation « Rechercher un article », il est stipulé que l'on veut faire cette recherche à partir d'une chaîne de caractères saisie par l'internaute. La recherche ne doit pas tenir compte des minuscules ou majuscules, et doit porter sur le nom de l'article ou le nom du produit. Voici cette requête :

Requête pour rechercher les articles

```
Query query = em.createQuery("SELECT i FROM Item i WHERE ❶
    UPPER(i.name) LIKE :keyword ❷
    OR UPPER(i.product.name) LIKE :keyword ❸
    ORDER BY i.product.category.name, i.product.name"); ❹
query.setParameter("keyword", "%" + keyword.toUpperCase() + "%"); ❺
List<Item> items = query.getResultList(); ❻
```

On sélectionne les articles ❶ dont le nom ❷, ou le nom du produit ❸, ressemble (LIKE) à la chaîne de caractères passée en paramètre ❺. On utilise la fonction UPPER pour mettre le résultat en majuscules et le comparer à la chaîne de caractères de recherche, elle aussi passée en majuscules (keyword.toUpperCase()). Le résultat est trié sur le nom de la catégorie puis le nom du produit ❹, et retourné via la méthode getResultList() ❻.

Démarcation de transactions

Comme nous venons de le voir, le stateless bean manipule les entités via l'entity manager (CRUD), et effectue des requêtes en base de données via JPQL. Toutes ces actions engendrent des insertions, des mises à jour et des suppressions de données qui doivent être cohérentes. C'est donc le rôle de l'EJB d'assurer une démarcation des transactions.

PERSISTENCE **Les jokers**

Tout comme SQL, JPQL peut aussi utiliser des « jokers » pour ses requêtes. Le joker '%' remplace n'importe quelle chaîne de caractères, y compris la chaîne vide. Le '_' remplace un et un seul caractère.

- ❖ On sélectionne les articles (Item).
- ❖ On utilise la fonction UPPER et le mot-clé LIKE.
- ❖ Pour obtenir le nom du produit, on appelle les accesseurs de l'objet : item.getProduct().getName().
- ❖ Le tri est fait sur le nom de la catégorie accessible par item.getProduct().getCategory().getName().
- ❖ On concatène le caractère '%' à l'opérateur LIKE (par exemple, LIKE '%iche%' récupère les caniches).
- ❖ L'exécution de la requête retourne une liste.

PERSISTENCE **Commit et rollback**

En SQL, rollback signifie que l'on restaure les données d'une base à l'état où elles se trouvaient avant modifications. Ainsi, les dernières modifications sont annulées. À l'opposé, le commit permet de valider ces modifications, en laissant les données de la base dans un état cohérent.

Démarquer explicitement une transaction consiste à indiquer quand elle débute (begin) et quand elle se termine (commit ou rollback en cas d'échec).

Transactions

On parle de transaction lorsqu'un certain nombre de tâches doivent être effectuées ensemble. Par exemple, pour créer un bon de commande, plusieurs entités (bon de commande, ligne de commande, adresse de livraison) doivent être créés et donc insérés en base de données. Ces insertions doivent toutes être réalisées afin d'éviter les incohérences de données. En cas de problèmes d'accès aux données, on demande à la transaction de retrouver l'état initial des données en annulant toutes les mises à jour.

ACIDité

On attend d'une transaction les propriétés d'ACIDité, c'est-à-dire :

- **(A)tomicté** : une transaction doit être entièrement effectuée ou pas du tout. Par exemple, lors de la finalisation d'une commande, il est impératif que la mise à jour de l'adresse de livraison et la validation du paiement soient toutes les deux effectuées avec succès. Si une action échoue, alors l'autre ne doit pas être exécutée.
- **(C)ohérence** : la cohérence entre tables d'une même base doit être respectée, même en cas d'incident. Par exemple, les contraintes d'intégrité entre clés primaires et clés étrangères doivent être respectées.
- **(I)solation** si deux transactions T1 et T2 ont lieu simultanément, T1 ne doit pas voir les modifications de T2 tant que T2 n'a pas été committée, et inversement.
- **(D)urabilité** : lorsque la transaction est achevée, les modifications sont définitives.

► <http://java.sun.com/javaee/5/docs/tutorial/doc/bncih.html>

PERSISTANCE Transactions explicites

Si vous ne voulez pas laisser le conteneur gérer les transactions, vous pouvez le faire explicitement en utilisant JTA (*Java Transaction API*), spécification Java EE définissant un modèle de gestionnaire de transactions.

► <http://java.sun.com/products/jta/>

Avec Java EE, la gestion des transactions peut être déléguée au conteneur : on parle alors de *Container-Managed Transaction* (CMT). Le développeur n'a pas besoin d'utiliser d'API pour le faire. Il est cependant possible de déclarer explicitement le mode transactionnel lorsqu'il est souhaité (*Bean-Managed Transaction*, ou BMT).

Gestion des transactions par le conteneur

Lorsque la gestion des transactions est laissée au soin du conteneur, il n'est pas nécessaire d'écrire explicitement les begin, commit ou rollback. En annotant un stateless bean avec `@TransactionAttribute`, le conteneur est informé de la politique transactionnelle à utiliser. Si le conteneur ne rencontre pas d'exceptions, il effectuera lui-même le commit en respectant les paramètres de l'annotation, sinon, il lancera un rollback.

Prenons par exemple une méthode d'un stateless bean. À l'appel de cette méthode, une transaction démarre pour se terminer à la fin de l'exécution de cette même méthode. Que se passe-t-il maintenant si cette méthode appelle une autre méthode située dans un autre stateless bean ? Cette seconde méthode s'exécute-t-elle dans la même transaction que la

première ? Crée-t-on une nouvelle transaction ? Cela dépend de l'annotation `@javax.ejb.TransactionAttribute`, qui est utilisée pour définir la politique transactionnelle et peut prendre les six valeurs suivantes :

- **REQUIRED** : utilise la transaction si elle existe, sinon en démarre une nouvelle (valeur par défaut).
- **REQUIRES_NEW** : démarre toujours une nouvelle transaction (suspend celle en cours si elle existe).
- **MANDATORY** : nécessite une transaction. Si l'appelant n'a pas de transaction, une exception est levée.
- **NOT_SUPPORTED** : ne crée pas de transaction ni ne propage une transaction existante.
- **SUPPORTS** : ne démarre aucune transaction, utilise celle qui est en cours.
- **NEVER** : refuse toute transaction. Si l'appelant est dans une transaction, une exception est levée.

Exemple de stateless bean utilisant différents modes de transaction

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER) ❶

@TransactionAttribute(value=TransactionAttributeType.REQUIRED) ❷
public class CustomerBean implements CustomerRemote {
    @PersistenceContext(unitName = "petstorePU")
    private EntityManager em;

    public Customer createCustomer(Customer customer,
                                   Address homeAddress) {

        customer.setHomeAddress(homeAddress);
        em.persist(customer);
        return customer;
    }

    @TransactionAttribute(value = TransactionAttributeType.NEVER) ❸
    public Customer findCustomer(final Long customerId) {

        Customer customer = em.find(Customer.class, customerId);
        return customer;
    }
}
```

APPROFONDIR Modes de transactions

Les différents modes de transactions sont détaillés sur le site de Sun :

► <http://java.sun.com/javaee/5/docs/tutorial/doc/bncih.html>

Par défaut, la gestion des transactions est gérée par le conteneur (CONTAINER). Pour qu'elle soit gérée explicitement par le développeur, il faut utiliser BEAN.

Placée sur l'EJB, cette annotation indique au conteneur que toutes les méthodes utilisent le mode transactionnel REQUIRED.

Cette méthode est en mode REQUIRED.

Cette méthode redéfinit sa police transactionnelle. Elle ne tient pas compte du mode REQUIRED mais utilise NEVER.

L'EJB `CustomerBean` commence par indiquer que les transactions doivent être gérées par le conteneur ❶ (CMT). Ensuite, comme on peut le voir dans le code précédent, l'annotation `TransactionAttribute` peut être utilisée à deux emplacements :

- Sur la classe de l'EJB ❷ : l'annotation définit alors le mode transactionnel à appliquer sur toutes les méthodes de l'EJB.
- Sur une méthode ❸ : elle spécifie ce mode transactionnel pour une seule méthode, ne tenant pas compte de l'annotation mère.

PERSISTENCE

Transactions dans YAPS Pet Store

L'application YAPS Pet Store n'a pas de spécificités transactionnelles. Les stateless session beans utilisent donc la valeur par défaut, c'est-à-dire le mode REQUIRED.

Gestion des exceptions

Java propose un mécanisme de gestion d'exceptions, consistant à effectuer les instructions dans un bloc d'essai (le bloc `try`) qui surveille les instructions. Lors de l'apparition d'une erreur, celle-ci est interceptée dans un bloc de traitement d'erreurs (le bloc `catch`) sous forme d'un objet appelé `Exception`. Ce bloc peut alors traiter l'erreur ou la relancer vers un bloc de plus haut niveau.

Ce mécanisme de gestion d'exceptions du langage Java est utilisé par les `stateless beans`. Par contre, la nouvelle spécification des EJB 3 distingue les exceptions applicatives des exceptions système.

Les exceptions

En Java, les exceptions sont des objets, instances de `java.lang.Throwable`. Elles peuvent être répertoriées en deux catégories : exception contrôlée et exception non contrôlée.

- Les exceptions non contrôlées (*unchecked exceptions*) sont, en général, lancées par le système. Elles correspondent à des erreurs à l'exécution et proviennent d'extensions de la classe `java.lang.RuntimeException`. Le compilateur java n'exige pas qu'elles soient déclarées ou traitées par les méthodes qui peuvent les lancer.
- Les exceptions contrôlées (*checked exceptions*) correspondent à des exceptions créées par l'utilisateur et proviennent d'extensions de la classe `java.lang.Exception`. Le compilateur exige qu'une méthode, dans laquelle une telle exception est lancée, déclare cette exception dans sa signature, ou bien la traite.

Exceptions d'application

Les EJB utilisent les exceptions d'application pour informer un client qu'une anomalie applicative s'est produite (paramètre invalide, numéro de carte de crédit erroné, etc.). Ce type d'exception n'est pas fait pour remonter les problèmes système (base de données indisponible, file d'attente JMS introuvable, etc.).

Une exception d'application peut soit être contrôlée (hérite de la classe `java.lang.Exception`), soit non contrôlée (hérite alors de la classe `java.lang.RuntimeException`). En revanche, elle ne doit hériter ni de `java.rmi.RemoteException` ni de `javax.ejb.EJBException`, qui sont réservées aux exceptions système. Sa seule particularité est qu'elle doit être annotée par `@ApplicationException`.

Lorsqu'une exception d'application est lancée par une méthode d'EJB, le conteneur l'intercepte et peut alors décider de « rollbacker » la transaction ou non. Ce choix est laissé au développeur. En effet, l'annotation

`@ApplicationException` possède un attribut `rollback` qui, positionné à `true`, informe le conteneur de l'annulation de la transaction.

Code de l'annotation `@javax.ejb.ApplicationException`

```
package javax.ejb;

@Target(value = {TYPE}) @Retention(value = RUNTIME)
public @interface ApplicationException {
    boolean rollback() default false;
}
```

◀ Cette annotation s'applique à une classe.

◀ Spécifie si le conteneur doit ou non rollbacker la transaction en cours. Par défaut, la valeur est `false`.

L'application YAPS Pet Store utilise certaines de ces exceptions applicatives, par exemple, pour la validation des cartes de crédit. Lorsque le client désire acheter des animaux domestiques, il doit saisir son numéro de carte. Ce dernier est analysé par le service web de la banque Bark-Bank (détaillé dans le chapitre 9, *Échanges B2B*). Si la carte est refusée, l'application lance une `CreditCardException` et il faut alors « rollbacker » les mises à jour qui auraient pu être faites dans la base de données.

Exception applicative `CreditCardException`

```
@ApplicationException(rollback = true)
public class CreditCardException extends RuntimeException {

    public CreditCardException(String message) {
        super(message);
    }
}
```

Dans le chapitre précédent, *Objets persistants*, nous avons vu que les entités validaient leurs attributs avant d'être persistés. Si une valeur est invalide, une `ValidationException` est lancée avec le paramètre `rollback` à `true`.

Entity Category validant ses attributs

```
@Entity
public class Category
    (...)
    @PrePersist
    @PreUpdate
    private void validateData() {
        if (name == null || "".equals(name))
            throw new ValidationException("Invalid name");
        if (description == null || "".equals(description))
            throw new ValidationException("Invalid description");
    }
}
```

Exception applicative ValidationException

```
@ApplicationException(rollback = true)
public class ValidationException extends RuntimeException {

    public ValidationException(String message) {
        super(message);
    }
}
```

Exception système

Une exception système est lancée lorsqu'une ressource système dysfonctionne (base de données indisponible, etc.). Ces exceptions ne sont pas attendues par l'application qui, la plupart du temps, ne sait pas comment les traiter. On peut donc laisser ces exceptions se propager pour enfin être interceptées par le conteneur. Le conteneur effectuera automatiquement un rollback sur la transaction et détruira le stateless bean.

Les exceptions de type système héritent de `java.rmi.RemoteException`, `javax.ejb.EJBException` ou de `java.lang.RuntimeException` (mais sans utiliser l'annotation `@ApplicationException`). Il est important de comprendre que des exceptions non contrôlées peuvent être lancées par d'autres systèmes et qu'elles n'ont rien à voir avec la modélisation d'un EJB métier. Elles ne doivent donc généralement pas apparaître dans la signature des méthodes. En revanche, pour les exceptions contrôlées (telles que `javax.jms.JMSException`), l'EJB est obligé de les « catcher ». Une bonne pratique consiste à encapsuler ce type d'exception dans une `EJBException` et la relancer. Le conteneur pourra alors la traiter comme une exception système et appliquer la politique transactionnelle.

Exemple de traitement d'exception JMS

```
public void publishOrder(Order order) {
    try {
        Session session = connection.createSession(true, 0);
        (...)
        ObjectMessage objectMessage = session.createObjectMessage();
        objectMessage.setObject(order);
        producer.send(objectMessage);
    } catch (JMSException e) {
        throw new EJBException(e);
    }
}
```

Le cycle de vie d'un stateless bean

Le cycle de vie d'un stateless bean est très simple. En fait, il n'a que deux états : soit il existe, soit il n'existe pas. L'état inexistant signifie que l'EJB n'a pas encore été instancié par le conteneur et n'existe pas en mémoire. Le passage à l'état prêt se fait lorsque le client invoque un EJB. Le conteneur crée alors une nouvelle instance et appelle la méthode demandée.

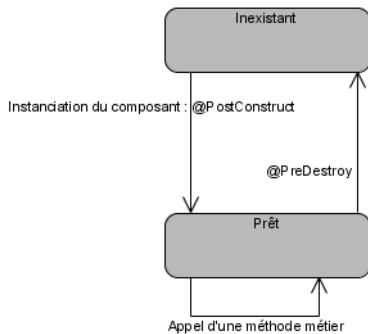


Figure 5–2
Cycle de vie d'un stateless bean

Le passage d'un état à l'autre peut être intercepté grâce aux annotations de callback.

Les annotations de callback

Grâce aux annotations de callback, le conteneur d'EJB laisse la possibilité aux développeurs d'effectuer des traitements lors du passage d'un état à un autre. Il existe deux annotations utilisables par les stateless beans :

- `@javax.annotation.PostConstruct`
- `@javax.annotation.PreDestroy`

Après avoir instancié un stateless bean, le conteneur exécute les méthodes annotées par `@PostConstruct`. Dans le cas inverse, l'annotation `@PreDestroy` est appelée lorsque le conteneur supprime l'EJB de la mémoire. Ces annotations sont importantes lorsque, par exemple, un EJB utilise des ressources externes. Si l'initialisation et la libération de ces ressources sont coûteuses, il est préférable de les effectuer le moins souvent possible, c'est-à-dire une fois à la création de l'EJB, et une fois à sa suppression.

Dans l'exemple ci-après, le stateless bean ouvre une connexion JMS dans la méthode `openConnection` (annotée par `@PostConstruct`) et la referme avant la suppression de l'EJB de la mémoire (`@PreDestroy`). Ainsi, tout au long de la période où le stateless bean est dans l'état prêt, la connexion reste ouverte et utilisable.

EJB Les stateless stockés dans un pool

Bien que les spécifications n'obligent pas les conteneurs à avoir un pool de stateless beans, la plupart des serveurs d'applications en utilise un pour augmenter les performances. Cela évite de créer et de détruire un EJB à chaque appel, puisque le conteneur n'a qu'à piocher dans son pool pour réutiliser le premier stateless disponible. Au démarrage, le conteneur crée un certain nombre de stateless beans et les stocke dans un pool. Si, à un moment donné, aucun n'est disponible, le conteneur en instancie de nouveaux pour effectuer des traitements. Au contraire, lorsque le conteneur a besoin de libérer de la mémoire, il supprime certains EJB du pool. GlassFish utilise ce mécanisme de pool. La taille de ce pool est paramétrable, tout comme sa taille minimale et maximale. Cette configuration est faite via la console d'administration (menu *Configuration>EJB Container>EJB Settings*).

SYNTAXE Callback

Une méthode annotée par une annotation callback respecte les règles suivantes :

- elle peut être annotée par une ou plusieurs annotations de callback ;
- elle ne peut lancer que des `RuntimeException` ;
- elle peut être `public`, `package`, `protected` ou `private`.

REMARQUE Utilisation de JMS

Ne vous inquiétez pas si vous ne comprenez pas totalement le code d'appel à JMS. Les traitements asynchrones seront étudiés de manière approfondie au chapitre 10, *Traitements asynchrones*.

Le stateless bean utilise une connexion JMS.

La connexion à JMS est ouverte après l'instanciation de l'EJB par le conteneur. L'annotation `@PostConstruct` intercepte cet événement et ouvre la connexion à JMS.

À la destruction de l'EJB, le conteneur appelle cette méthode qui referme la connexion à JMS.

Stateless bean utilisant des annotations de callback

```
@Stateless(mappedName = "ejb/stateless/Order")
public class OrderBean implements OrderRemote, OrderLocal {

    private ConnectionFactory connectionFactory;
    private Connection connection;
    (...)

    @PostConstruct
    public void openConnection() {
        connection = connectionFactory.createConnection();
    }

    @PreDestroy
    public void closeConnection() {
        if (connection != null) {
            connection.close();
        }
    }
}
```

Les stateless beans de YAPS Pet Store

Maintenant que nous avons vu le fonctionnement des EJB sans état, voyons comment les utiliser dans notre application YAPS Pet Store.

Contrairement aux entités qui représentent les objets métier (identifiables par les mots des cas d'utilisation), les stateless beans représentent les actions. Ils se rapportent aux verbes des cas d'utilisation. Par exemple, créer un client, mettre à jour un client, rechercher un article, supprimer un bon de commande, etc.

Un EJB ne représente pas une seule de ces actions, mais plusieurs regroupées au sein d'une même classe. L'attribution de ces actions aux bonnes classes est l'un des problèmes de la conception orientée objet. Pour chaque action, il nous faut décider dans quelle classe la mettre. Une erreur fréquemment rencontrée consiste à définir un stateless bean pour chaque entité. Cette relation un pour un peut exister, mais ne doit pas être la règle.

L'application YAPS Pet Store utilise trois stateless beans :

- `CatalogBean` pour la gestion du catalogue au sens large, c'est-à-dire les catégories, les produits et les articles ;
- `CustomerBean` pour la création, mise à jour, suppression des clients ainsi que leur connexion au site marchand ;
- `OrderBean` pour les bons de commande, les lignes de commandes et tout ce qui concerne le paiement par carte de crédit (communication avec `BarkBank`) ainsi que l'acheminement (`PetEx`).

La gestion des clients

Si on se reporte au cas d'utilisation « Gérer les clients », on s'aperçoit que les employés de la société doivent avoir la possibilité de créer, mettre à jour, supprimer et lister les clients de l'application. Leur interface graphique (Swing) étant distante, elle utilisera les méthodes de l'interface `CustomerRemote`.

Les clients, quant à eux, utilisent l'interface web pour se connecter au système, consulter et mettre à jour leurs coordonnées (cas d'utilisation « Consulter et modifier son compte »). L'application web étant déployée dans le même ear que les stateless beans, elle utilise l'interface locale (`CustomerLocal`).

Ci-après, le diagramme de classes représentant ces deux interfaces ainsi que la classe d'implémentation (`CustomerBean`). Celle-ci utilise les entités `Customer` et `Address`.

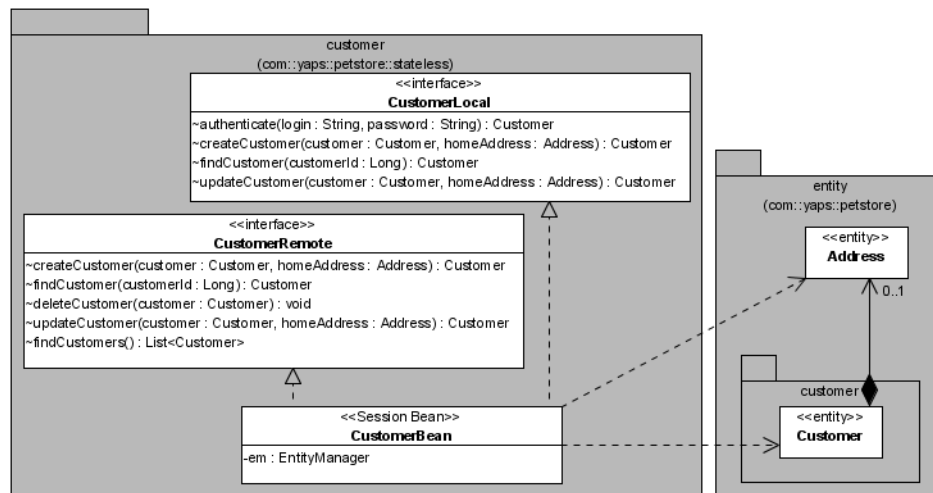


Figure 5-3
Diagramme de classes
du stateless bean `Customer`

Notez les différences entre les deux interfaces : l'interface `CustomerLocal` permet l'authentification (en effet, seule l'interface web le permet) alors que l'interface distante permet aux employés d'afficher la liste de tous les clients et d'en supprimer.

CustomerLocal

Les méthodes référencées dans l'interface locale sont utilisables par l'interface web. Elles permettent au client de se créer un compte, de s'authentifier, de consulter et de mettre à jour ses informations.

UML Les classes

En UML, une classe est représentée par un rectangle, séparé en trois parties :

- la première partie contient le nom de la classe et un éventuel stéréotype ;
- la deuxième contient les attributs de la classe ;
- la dernière contient les méthodes de la classe.

On peut masquer ou non une de ces parties si on veut rendre un diagramme plus lisible ou si la classe en question ne contient ni attribut ni méthode.

Interface locale.



```
@Local
public interface CustomerLocal {
    Customer authenticate(String login, String password);
    Customer createCustomer(Customer customer, Address address);
    Customer findCustomer(Long customerId);
    Customer updateCustomer(Customer customer,Address address);
}
```

Méthodes utilisées par l’application web.



```
Customer authenticate(String login, String password);
Customer createCustomer(Customer customer, Address address);
Customer findCustomer(Long customerId);
Customer updateCustomer(Customer customer,Address address);
}
```

CustomerRemote

Les méthodes de l’interface distante permettent aux employés de gérer les clients du système.

Interface distante.



```
@Remote
public interface CustomerRemote {
```

Méthodes utilisées par l’application en client lourd.



```
Customer createCustomer(Customer customer, Address address) ;
Customer findCustomer(Long customerId) ;
void deleteCustomer(Customer customer) ;
Customer updateCustomer(Customer customer, Address address) ;
List<Customer> findCustomers() ;
}
```

CustomerBean

La classe d’implémentation du stateless bean manipule les entités Customer et Address via l’entity manager et JPQL.

La politique transactionnelle est celle par défaut.



```
@TransactionAttribute(value=
TransactionAttributeType.REQUIRED)
```

L’annotation Stateless identifie cette classe comme étant un stateless bean qui se nomme CustomerSB.



```
@Stateless(name = "CustomerSB",
mappedName = "ejb/stateless/Customer")
```

Le bean implémente l’interface locale et distante.



```
public class CustomerBean implements CustomerRemote,
CustomerLocal {
```

L’entity manager est lié au contexte de persistance “petstorePU” qui est défini dans le fichier persistence.xml.



```
@PersistenceContext(unitName = "petstorePU")
private EntityManager em;
```

Cette méthode permet au client de s’authentifier à l’aide de son login et mot de passe.



```
public Customer authenticate(String login, String password) {
    if (login == null || "".equals(login))
        throw new ValidationException("Invalid login");
```

On s’assure de la validité du paramètre login en lançant une exception si besoin.



```
if (login == null || "".equals(login))
    throw new ValidationException("Invalid login");
```

On recherche l’entity Customer à partir de son login grâce à une requête JPQL.



```
Query query;
Customer customer;
```

<pre> query = em.createQuery("SELECT c FROM Customer c WHERE c.login=:login"); query.setParameter("login", login); customer = (Customer) query.getSingleResult(); if (customer != null) customer.matchPassword(password); return customer; } </pre>	◀ Si la recherche aboutit (l'objet <code>customer</code> est donc différent de <code>null</code>), on appelle la méthode métier <code>matchPassword</code> de l'entity. ◀ On retourne l'entity s'il a été trouvé. Sinon, cette méthode renvoie la valeur <code>null</code>.
<pre> public Customer createCustomer(Customer customer, Address homeAddress) { if (customer == null) throw new ValidationException("Customer object is null"); customer.setHomeAddress(homeAddress); em.persist(customer); return customer; } </pre>	◀ On crée un client à partir des entités <code>Customer</code> et <code>Address</code>. ◀ On s'assure que l'entity <code>Customer</code> existe. En revanche, un client peut ne pas fournir d'adresse. ◀ On relie les entités entre eux. ◀ L'entity manager persiste le client dans la base de données.
<pre> public Customer findCustomer(Long customerId) { if (customerId == null) throw new ValidationException("Invalid id"); Customer customer; customer = em.find(Customer.class, customerId); return customer; } </pre>	◀ À partir de son identifiant, cette méthode retourne un entity <code>Customer</code>.
<pre> public void deleteCustomer(Customer customer) { if (customer == null) throw new ValidationException("Customer object is null"); em.remove(em.merge(customer)); } </pre>	◀ Cette méthode supprime le client passé en paramètre. ◀ L'entity doit être rattaché (<code>merge</code>) avant de pouvoir être supprimé.
<pre> public Customer updateCustomer(Customer customer, Address homeAddress) { if (customer == null) throw new ValidationException("Customer object is null"); customer.setHomeAddress(homeAddress); em.merge(customer); } </pre>	◀ Cette méthode met à jour les données du client. ◀ L'appel au <code>merge</code> rattache l'entity au manager, et synchronise les éventuelles mises à jour.

- Cette méthode retourne la liste de tous les clients.
- Grâce à une query JPQL, tous les entités Customer sont ramenés de la base de données.

```
return customer;
}

public List<Customer> findCustomers() {

    Query query;
    List<Customer> customers;

    query = em.createQuery("SELECT c FROM Customer c");
    customers = query.getResultList();

    return customers;
}
}
```

UML Les différents liens

En UML, les diagrammes de classes utilisent toutes sortes de liens. L'image suivante représente, de gauche à droite, un héritage, une implémentation, une utilisation et une association.



Figure 5-4 Liens UML

UML La visibilité des attributs et méthodes

Les méthodes et les attributs d'une classe utilisent différents modes de visibilité. Ceux-ci ont une représentation graphique en UML : privée (-), protégée (#), paquetage (~) et publique (+).



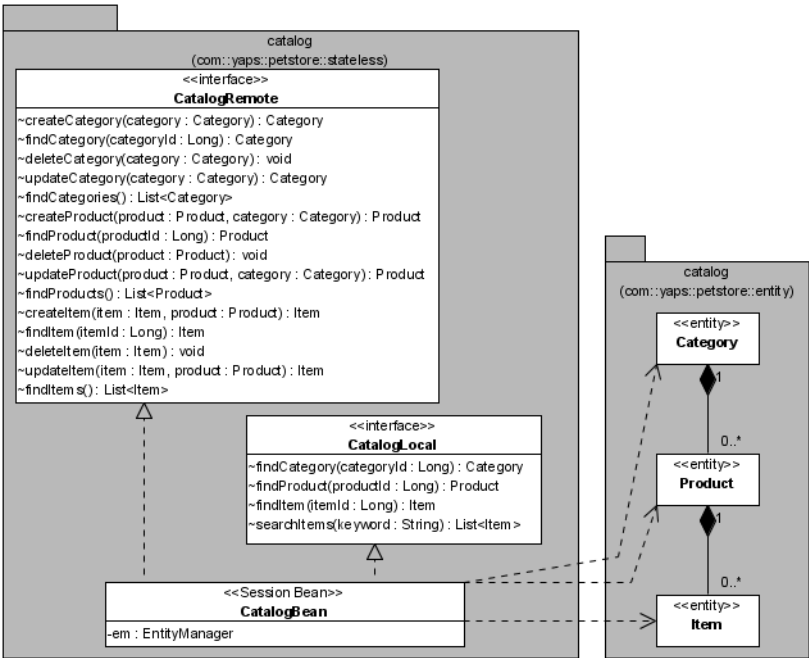
Figure 5-5 Visibilit   en UML

Figure 5-6
Classe et interfaces
du stateless bean catalogue

La gestion du catalogue

Le catalogue de YAPS est organis   en cat  gories, produits et articles. Pour pouvoir mettre    jour le catalogue, l'application riche (Swing) doit permettre aux employ  s de mettre    jour chacun de ces   l  ments. On retrouve ainsi des m  thodes CRUD pour la cat  gorie, le produit et l'article dans l'interface distante CatalogRemote.

Le site web, quant    lui, ne permet pas les mises    jour, mais simplement la consultation du catalogue et la recherche d'articles (interface CatalogLocal) par les clients et les internautes.



Le stateless bean `CatalogBean` manipule les entités `Category`, `Product` et `Item`.

CatalogBean

Vous l'aurez compris, le code des méthodes CRUD des éléments du catalogue, ressemble de très près à celui du client que nous venons de voir. Toute description de ce code est donc superflue. L'extrait de code ci-après nous montre uniquement la méthode de recherche des articles.

```
@TransactionAttribute(value=
TransactionAttributeType.REQUIRED)
@Stateless(name = "CatalogSB",
mappedName = "ejb/stateless/Catalog")

public class CatalogBean implements
CatalogRemote, CatalogLocal{

    @PersistenceContext(unitName = "petstorePU")
    private EntityManager em;

    (...)

    public List<Item> searchItems(String keyword) {
        Query query;
        List<Item> items;

        query = em.createQuery("SELECT i FROM Item i WHERE
            UPPER(i.name) LIKE :keyword OR
            UPPER(i.product.name) LIKE :keyword
            ORDER BY i.product.category.name, i.product.name");

        query.setParameter("keyword",
            "%" + keyword.toUpperCase() + "%");
        items = query.getResultList();

        return items;
    }
}
```

◀ Le nom JNDI de l'EJB est "ejb/stateless/Catalog".

◀ Les autres méthodes ne sont pas décrites.

◀ Cette méthode retrouve tous les articles dont le nom correspond au mot-clé passé en paramètre.

La gestion des bons de commande

Les bons de commande ne sont pas automatiquement créés par une interface graphique. C'est le processus d'achat d'animaux par le client qui déclenche la création d'un bon de commande et de toutes les actions qui en découlent (envoyer un e-mail de confirmation au client, avertir le transporteur, etc.).

Ajout de trace

La journalisation consiste à garder les traces des événements survenus dans une application. Des fichiers de log au format prédéfini conservent des messages informant sur la date et l'heure de l'événement, sa nature, sa gravité, une description et d'autres informations : utilisateur, classe, etc. L'API `java.util.logging`, fournie par défaut depuis le JDK 1.4, permet de journaliser des événements dans un fichier texte ou XML, et utilise différents niveaux de sévérité.

Dans l'architecture du YAPS Pet Store, toutes les actions passent par la couche de stateless beans (la façade). C'est donc l'endroit idéal pour rajouter des traces dans le code. En utilisant l'API de logging, on peut ainsi délimiter l'entrée (entering) et la sortie (exiting) d'une méthode, ou bien rajouter des traces au milieu du code.

Ci-après un exemple de traces dans l'EJB OrderBean.

```
public class OrderBean implements OrderRemote, OrderLocal {
    private Logger logger = Logger.getLogger(
        "com.yaps.petstore.stateless"); ❶

    private final String cname = this.getClass().getName();
    (...)
    public List<Order> findOrders() {
        final String mname = "findOrders";
        logger.entering(cname, mname); ❷

        Query query;
        List<Order> orders;

        logger.finest("Recherche les bons de commande"); ❸

        query = em.createQuery("SELECT o FROM Order o");
        orders = query.getResultList();

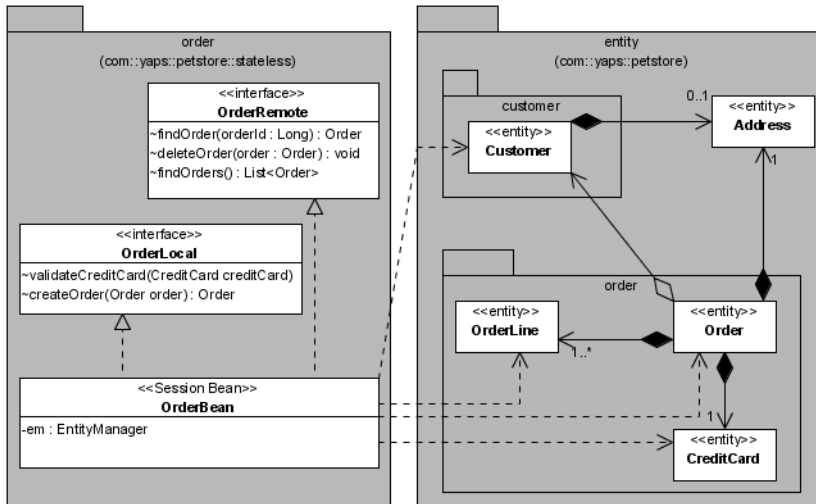
        logger.exiting(cname, mname, orders.size()); ❹
        return orders;
    }
}
```

Un logger ❶ portant le nom de `com.yaps.petstore.stateless` est défini. Il est ensuite utilisé pour tracer l'entrée ❷ et la sortie ❹ de la méthode, tout comme pour ajouter des traces dans le code ❸.

Il faut cependant faire attention au nombre de traces que vous ajoutez dans le code ainsi qu'à leur niveau de sévérité. Il faut garder en tête que, potentiellement, une journalisation se solde par un accès disque pour écrire dans un fichier. Cela peut donc avoir des répercussions en termes de performances. Dans ce cas, veillez à profiter des différents niveaux de criticité pour renseigner à bon escient vos traces et définir correctement ce degré pour limiter le stockage sur disque.

- <http://www.onjava.com/pub/a/onjava/2002/06/19/log.html>
- <http://java.sun.com/j2se/1.5.0/docs/api/java/util/logging/package-summary.html>

Ci-après le diagramme de classes du stateless bean agissant sur le bon de commande.



Les méthodes du bon de commande étant étroitement liées au panier électronique, leurs implémentations seront décrites dans le chapitre 8, intitulé *Gestion du panier électronique*.

Paquetages des stateless beans

Les interfaces et classes des stateless beans (client, catalogue et bon de commande) sont placées dans les sous-paquetages de `com.yaps.petstore.stateless`. Les exceptions lancées dans les EJB (`ValidationException` et `CreditCardException`), se trouvent dans `com.yaps.petstore.exception`.

Architecture

L'architecture globale obtenue jusqu'à présent, est principalement constituée de stateless et d'entités. La couche de stateless forme le point d'entrée, c'est-à-dire la façade de l'application et publie des méthodes pour les interfaces graphiques. Cette couche s'appuie ensuite sur les entités qui représentent le modèle persistant de l'application.

La couche des stateless beans est constituée de trois EJB permettant la gestion du catalogue (`CatalogBean`), des clients (`CustomerBean`) et des bons de commande (`OrderBean`). Chacun publie ses méthodes à l'aide d'interfaces locales et distantes. Les stateless beans interagissent avec les entités.

UML Les stéréotypes

Les diagrammes de classes s'enrichissent ainsi que la palette de stéréotypes. `<<interface>>` permet de typer une classe en tant qu'interface (en UML on peut aussi utiliser le rond pour représenter une interface), `<<Session Bean>>` et `<<entity>>` sont utilisés pour les EJB.

Figure 5-7

Diagramme des classes interagissant avec la gestion du bon de commande

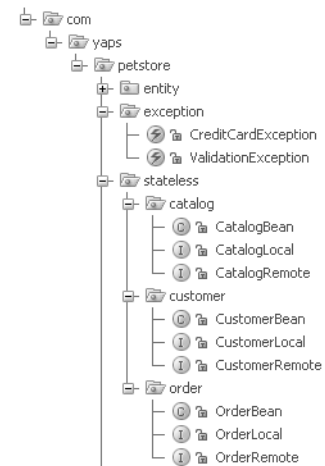


Figure 5-8 Stateless beans et exceptions

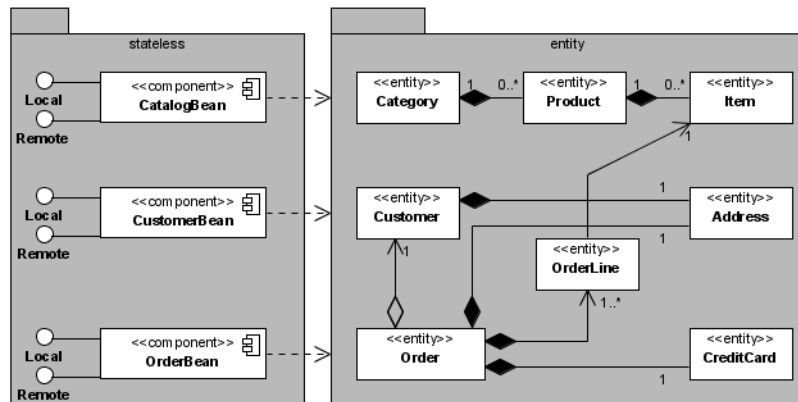
UML Les composants

Les composants, en UML, sont représentés par le stéréotype `<<component>>`. Ils publient une ou plusieurs interfaces représentées par un cercle.



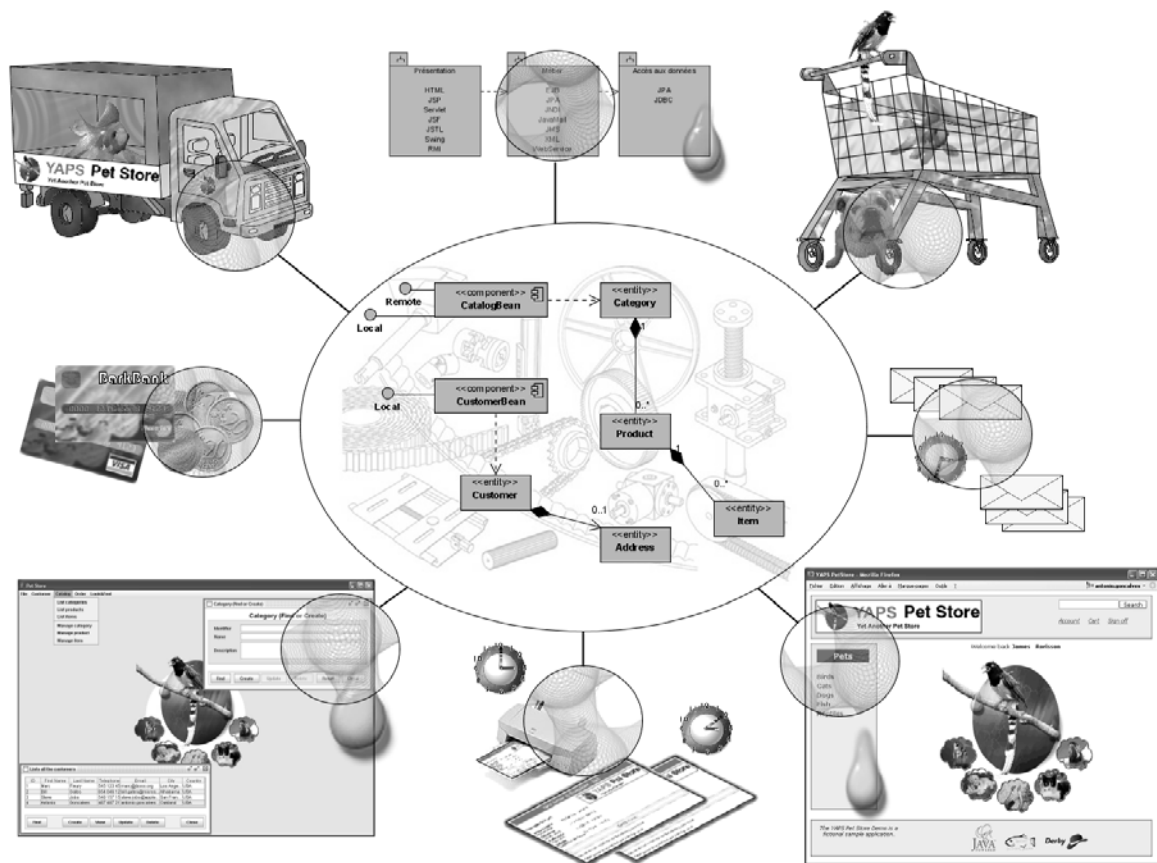
Figure 5-9 Composant et interface

Figure 5–10
Architecture globale du YAPS Pet Store



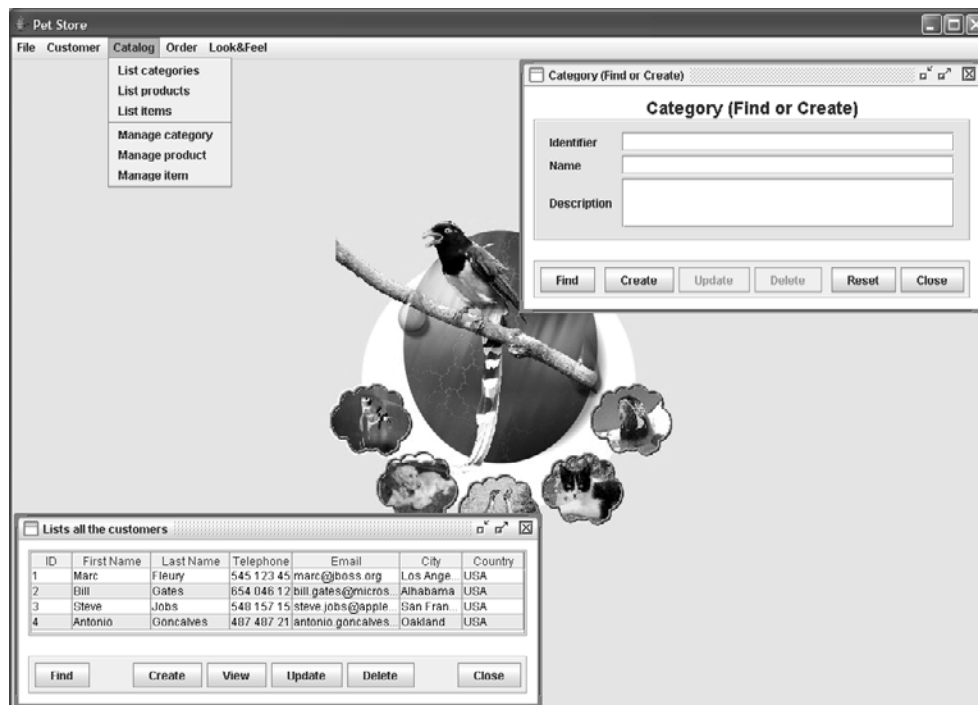
En résumé

Les stateless beans forment une couche de traitements métier qui sera invoquée par les interfaces graphiques. Ils manipulent les entités, assurent la cohérence des données grâce aux transactions, et sont accessibles de manière locale ou distante. La spécification EJB 3 simplifie grandement leur développement via l'utilisation des annotations. Pièce incontournable de Java EE, les EJB seront déployés sur le serveur GlassFish dans le prochain chapitre et seront utilisés à l'aide d'une application distante Swing.



6

chapitre



Exécution de l'application

Les couches de persistance et de traitements métier ont été développées dans les précédents chapitres. Il reste maintenant à y accéder au travers d'une application cliente. Ce chapitre nous explique comment compiler l'application YAPS Pet Store, la packager et la déployer sur le serveur GlassFish. L'interface utilisateur Swing accédera de manière distante aux stateless beans en utilisant JNDI.

SOMMAIRE

- ▶ Couche de présentation
- ▶ Client Swing
- ▶ Appel distant des stateless beans
- ▶ Business Delegate et Service Locator
- ▶ Compiler, packager, déployer
- ▶ Exécuter l'application

MOTS-CLÉS

- ▶ Swing
- ▶ JNDI
- ▶ GlassFish
- ▶ Déploiement
- ▶ Ant

APPROFONDIR Swing

Le rôle de ce livre n'est pas de couvrir l'API Swing. Pour de plus amples informations, reportez-vous aux références suivantes :

Creating a GUI with JFC/Swing (The Swing Tutorial)

► <http://java.sun.com/docs/books/tutorial/uiswing>

Java CodeGuru: Swing

► <http://www.codeguru.com/java/Swing/index.shtml>

📖 Kathy Walrath, Mary Campione, Alison Huml, Sharon Zakhour, *The JFC Swing Tutorial: A Guide to Constructing GUIs, Second Edition*, Addison-Wesley, 2004

📖 Emmanuel Puybaret, *Swing*, Eyrolles, 2006

📖 Jean-Baptiste Briaud, Valérie Berthié, *Swing – la synthèse : développement des interfaces graphiques en Java*, Dunod, 2005

REMERCIEMENT L'application Swing Pet Store

Je tiens à remercier David Dewalle pour m'avoir aidé à développer l'application graphique de YAPS Pet Store. Celle-ci utilise entre autres le framework Open Source Greenauks dont David est le créateur et qui simplifie le développement MVC et la gestion des événements Swing.

► <http://greenauks.v5projects.org/>

Les objets persistants et la couche métier ont été développés dans les précédents chapitres. Ces composants serveurs doivent maintenant être compilés puis déployés sur GlassFish pour pouvoir être interrogés par l'application graphique. Ce chapitre se consacre donc à l'interface homme-machine (IHM) et aux processus de déploiement.

Les employés de la société gèrent le catalogue des articles, les clients et affichent les bons de commande à l'aide d'une interface graphique de type client lourd (voir les cas d'utilisation « Gérer les clients », « Gérer le catalogue » et « Visualiser et supprimer les commandes »). Développée en Swing, cette interface doit être installée et exécutée sur le poste de chaque employé.

Swing

La langage Java dispose de différentes API permettant de construire des interfaces graphiques sophistiquées telles que AWT et Swing. AWT est antérieure à Swing et fut développée pour la première version du JDK (1.0) alors que Swing est apparue en tant que librairie annexe dans cette même version mais n'a été intégrée dans le JDK qu'à partir de la version 1.2 (soit Java 2). Il en résulte donc des différences fondamentales de conception entre les deux librairies.

Par exemple, un composant AWT est associé à une fenêtre gérée par le système d'exploitation sous-jacent, responsable de son apparence. Par opposition, les composants Swing sont simplement dessinés à l'intérieur de leur conteneur comme s'il s'agissait d'une image. Le système d'exploitation n'est pas sollicité mais uniquement la machine virtuelle Java. Avec Swing, les possibilités graphiques sont décuplées. On peut par exemple créer des boutons comportant une image à la place d'un texte, des boutons ronds, des bordures variées pour les composants, ou encore utiliser un composant d'arborescence (JTree).

Ce chapitre ne couvre pas l'API Swing car elle est trop riche et ne correspond pas au thème principal de ce livre, qui n'est autre que Java EE. Swing n'est utilisée ici que comme support graphique pour appeler des EJB de manière distante via JNDI.

Exemple d'appel à un EJB dans Swing

Dans la gestion du catalogue, les employés peuvent consulter le détail d'une catégorie à partir d'un écran Swing. Un extrait du code ci-après nous montre les étapes nécessaires à l'appel de la méthode `findCategory` de l'EJB `stateless CatalogBean`.

Écran d'affichage de la catégorie

```
public class CategoryCrudFrame extends JInternalFrame {
    (...)
    public void findActionPerformed(EventObject evt) {

        Context initialContext = new InitialContext(); ❶
        CatalogRemote catalogRemote = (CatalogRemote)
            initialContext.lookup("ejb/stateless/Catalog"); ❷

        Category category =
            catalogRemote.findCategory(identifiant); ❸

        model.setIdentifier(category.getId());
        model.setName(category.getName()); ❹
        model.setDescription(category.getDescription());
    }
}
```

L'application distante a tout d'abord besoin d'obtenir le contexte initial JNDI ❶. À partir de ce contexte, on recherche (lookup) l'EJB qui gère le catalogue et dont le nom est `ejb/stateless/Catalog` ❷. On obtient ainsi l'interface distante sur laquelle on appelle la méthode métier du composant ❸. La méthode `findCategory` retourne un entity `Category` avec lequel on alimente les zones de l'écran ❹. Le résultat graphique est présenté sur l'écran ci-après (figure 6–1).

RAPPEL Le nom JNDI du CatalogBean

Dans cet exemple, le nom du stateless session bean est `ejb/stateless/Catalog`. Si vous vous reportez au code de l'EJB `CatalogBean`, vous verrez que ce nom est le même que celui défini dans l'annotation `@Stateless`.

```
@Stateless(mappedName=
    ➔ "ejb/stateless/Catalog")
```

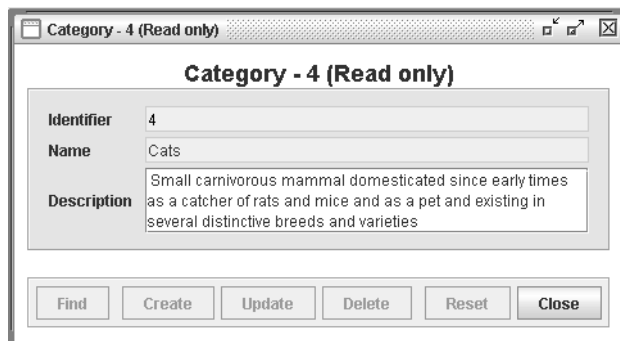


Figure 6–1
Affichage des informations d'une catégorie

JNDI

Dans tout système distribué, le service de nommage est un service fondamental. Il a pour vocation d'associer un nom à un objet et de permettre la recherche de cet objet à partir de son nom. Java Naming and Directory Interface (JNDI) fournit les fonctionnalités de nommage et d'annuaire aux applications écrites en Java. JNDI est une abstraction d'annuaire et peut donc être utilisé sur LDAP, NIS ou DNS par exemple. Ce service

APPROFONDIR JNDI

► <http://java.sun.com/products/jndi/>

LDAP, NIS et DNS

- **LDAP**, ou *Lightweight Directory Access Protocol*, est un protocole qui permet d'accéder à des annuaires répondant à la norme x500.
- **NIS** (*Network Information Service*) est le service qui permet à certaines informations d'être connues par toutes les machines disponibles sur un réseau.
- Un **DNS** (*Domain Name System*) est un serveur qui permet de faire correspondre une adresse web (URL) avec une adresse IP.

CONFIGURATION `jndi.properties`

Au lieu de coder en dur les paramètres d'accès à JNDI, on peut les externaliser dans un fichier nommé `jndi.properties`. Dans notre cas, nous allons déployer l'application sur le serveur GlassFish qui se trouve sur le même serveur physique (`localhost`) que l'interface cliente. Les paramètres par défaut du service JNDI n'ont donc pas besoin d'être modifiés. Nous utiliserons le fichier `jndi.properties` inclus dans la distribution GlassFish.

EJB **cast** ou **narrow** des interfaces ?

Pour ceux d'entre vous habitués aux EJB 2.x, remarquez qu'en EJB 3 le résultat du lookup peut être directement casté en interface remote sans avoir à utiliser la méthode `PortableRemoteObject.narrow()`.

permet à un client de localiser un objet ou une ressource distribuée. L'API JNDI est accessible à partir du paquetage `javax.naming.*`.

Dans l'exemple précédent, pour pouvoir afficher une catégorie, l'application Swing doit avant tout localiser l'interface `CatalogRemote` à partir d'un contexte initial. Dans ce type de système hiérarchique, un contexte peut être vu comme un nœud au sein d'un arbre. Le code `Context initialContext = new InitialContext()` permet d'obtenir la racine de cet arbre. Ensuite, charge à l'application de s'y déplacer pour obtenir l'objet qu'elle recherche. Par exemple le nom `ejb/stateless/Catalog` signifie qu'à partir de la racine il existe un contexte appelé `ejb`, puis un sous-contexte `stateless` dans lequel on trouve un objet nommé `Catalog`.

Cependant, pour retrouver ces objets, l'application doit connaître les paramètres d'accès au service JNDI. Ces paramètres sont propres à chaque serveur d'applications et peuvent, soit être mis dans un fichier externe (`jndi.properties`), soit directement dans le code.

Paramètres d'accès au service JNDI de GlassFish

```
public class CategoryCrudFrame extends JInternalFrame {
    (...)
    public void findActionPerformed(EventObject evt) {

        Properties props = new Properties(); ①
        props.setProperty("java.naming.factory.initial", ②
            "com.sun.enterprise.naming.SerialInitContextFactory");
        props.setProperty("java.naming.factory.url.pkgs", ②
            "com.sun.enterprise.naming");
        props.setProperty("java.naming.factory.state", ②
            "com.sun.corba.ee.impl.presentation.rmi.JNDIStateFactoryImpl");
        props.setProperty("java.naming.provider.url", "localhost");

        Context initialContext = new InitialContext(props); ③

        CatalogRemote catalogRemote = (CatalogRemote)
            initialContext.lookup("ejb/stateless/Catalog"); ④

        (...)
    }
}
```

Ce code crée un objet `Properties` ① dans lequel on affecte des valeurs spécifiques ② au serveur GlassFish afin d'accéder à son annuaire JNDI. Ces propriétés sont ensuite passées dans le constructeur du contexte initial ③ et permettent de retrouver l'objet recherché ④.

Application Client Container

Pour expliquer l'*Application Client Container* (ACC), il faut connaître le mécanisme d'injection, ou pattern IoC (*Inversion of Control*). Si ce n'est pas le cas, reportez-vous au chapitre suivant, *Interface web* où l'injection est expliquée en détail.

Reprenons notre exemple : une classe a besoin de contacter un EJB. Pour cela, elle utilise JNDI pour obtenir une référence vers cet EJB. Lorsque cette classe est exécutée dans un conteneur (EJB ou web), le conteneur lui-même injecte cette référence. Ce mécanisme permet à la classe de s'affranchir des appels JNDI.

Depuis la première version de J2EE, il existe un conteneur web pour exécuter et gérer le cycle de vie des servlets, et un conteneur EJB pour faire de même avec les Enterprise Java Beans. L'Application Client Container apporte les mêmes types de services, c'est-à-dire la sécurité, le service de nommage, l'injection, mais pour une application Java SE (Swing, dans notre exemple). Ainsi, en démarrant l'application à l'aide de la commande `%GLASSFISH_HOME%/bin/appclient`, nous aurions pu nous affranchir du Service Locator et bénéficier de l'injection offerte par l'ACC. Le code est alors allégé puisqu'il n'a plus besoin d'invoquer directement les classes de l'API JNDI.

Voici à quoi ressemble l'appel d'une méthode EJB lorsqu'on utilise l'Application Client Container :

```
public class CategoryCrudFrame extends
    JInternalFrame {
    (...)
    @EJB ①
    private CatalogRemote catalogRemote ②
    (...)
    public void findActionPerformed(EventObject evt) {
        Category category =
            catalogRemote.findCategory(identifiant); ③

        model.setIdentifiant(category.getId());
        model.setName(category.getName());
        model.setDescription(category.getDescription());
    }
}
```

Grâce à l'annotation `@EJB` ① (détaillée dans le chapitre suivant), l'ACC injecte une référence de l'interface distante de l'EJB dans la variable `catalogRemote` ②. Celle-ci peut ensuite être utilisée directement ③ sans passer par des appels explicites à JNDI.

Pour des raisons didactiques, j'ai préféré utiliser JNDI pour l'application cliente et l'injection pour les composants serveurs.

► <https://glassfish.dev.java.net/javaee5/docs/DG/beakt.html>

Comment développer l'application Swing

Nous n'aborderons pas les API Swing de l'application, mais nous nous intéresserons plutôt à l'architecture utilisée pour décorréliser la partie graphique des appels EJB. L'interface graphique utilisera donc les deux design patterns Business Delegate et Service Locator pour séparer la présentation de la logique de communication JNDI.

Service Locator

Comme nous l'avons vu précédemment dans le code, l'interface graphique a besoin de localiser les stateless beans au travers d'un service de nommage. Le code technique qui permet d'accéder à JNDI peut être isolé dans une même et seule classe : le Service Locator.

Le design pattern Service Locator, dont le seul but est de localiser des objets dans l'arbre JNDI, permet de rendre notre système plus flexible en centralisant et masquant les accès JNDI. En effet, au lieu d'ajouter ce code technique dans nos écrans Swing, ce design pattern fournit plusieurs méthodes pour retrouver des objets stockés dans le service de nommage.

APPROFONDIR Service Locator

Core J2EE Patterns – Service Locator

► java.sun.com/blueprints/corej2eepatterns/Patterns/ServiceLocator.html

Par exemple, l'extrait de code suivant décrit comment retrouver une interface distante. La méthode `getRemoteInterface` prend en paramètre le nom JNDI de la remote interface ❶, effectue un lookup JNDI ❷, puis retourne l'interface ❸ si elle l'a trouvé ou elle lance une exception de type `ServiceLocatorException` ❹ dans le cas contraire.

Extrait du code de la méthode récupérant les interfaces distantes

```
public Object getRemoteInterface(String jndiName) ❶
    throws ServiceLocatorException {
    Object remoteInterface;
    try {
        remoteInterface = initialContext.lookup(jndiName); ❷
    } catch (Exception e) {
        throw new ServiceLocatorException(e); ❹
    }
    return remoteInterface; ❸
}
```

L'autre avantage du Service Locator est d'optimiser la récupération des objets en utilisant un mécanisme de cache. Lorsqu'il trouve un objet dans JNDI, le Service Locator le stocke dans un cache (une `HashMap` par exemple), puis le réutilise directement lors des appels suivants. Ceci n'est possible que si le Service Locator implémente le design pattern Singleton.

Le singleton

Un singleton est une classe pour laquelle une et une seule instance existe dans toute l'application. Toutes les références d'objets de ce type sont en réalité des références à un même objet. Par conséquent, un singleton n'est stocké qu'une seule fois en mémoire.

L'utilisation d'un singleton réside dans la prévention de la création d'un objet autre que celui que vous fournissez. Pour ce faire, il suffit dans un premier temps de déclarer tous les constructeurs comme étant privés, et d'avoir une méthode publique et statique permettant de retourner l'instance unique du singleton.

► <http://c2.com/cgi/wiki?SingletonPattern>

Le code suivant vous montre le Service Locator implémenté en singleton et utilisant un système de cache (`private Map<String, Object> cache`).

Service Locator avec cache et singleton

```
public class ServiceLocator {
    private Context initialContext;
    private Map<String, Object> cache;
    private static ServiceLocator instance= new ServiceLocator();
```

Contexte initial JNDI.

Le cache est représenté par une `Map`.

Le design pattern Singleton utilise une méthode statique pour accéder à l'unique instance de l'objet.

```

public static ServiceLocator getInstance() {
    return instance;
}

private ServiceLocator() throws ServiceLocatorException {
    try {
        initialContext = new InitialContext();
        cache = new HashMap<String, Object>();
    } catch (Exception e) {
        throw new ServiceLocatorException(e);
    }
}

public Object getRemoteInterface(String jndiName)
    throws ServiceLocatorException {

    Object remoteInterface = cache.get(jndiName);

    if (remoteInterface == null) {
        try {
            remoteInterface = initialContext.lookup(jndiName);
            cache.put(jndiName, remoteInterface);
        } catch (Exception e) {
            throw new ServiceLocatorException(e);
        }
    }
    return remoteInterface;
}
}

```

◀ Le constructeur est privé, il ne peut donc pas être appelé par une classe extérieure, ce qui nous garantit l'unicité de l'instanciation. On initialise le contexte JNDI et le cache dans le constructeur.

◀ Méthode permettant de retrouver une interface distante.

◀ On commence par rechercher l'objet dans le cache.

◀ Si l'objet ne s'y trouve pas, on le recherche dans JNDI puis on le stocke dans le cache pour les utilisations futures.

Business Delegate

Nous venons d'isoler le code JNDI dans un Service Locator, il ne nous reste plus qu'à déléguer son appel par un Business Delegate. L'idée de ce pattern est d'avoir une classe Business Delegate par stateless session bean qui redéfinit chaque méthode distante. Les interfaces graphiques n'utilisent donc que les méthodes des objets Delegate sans se préoccuper de la façon de récupérer la référence sur l'interface distante. Ce design pattern permet de regrouper à un seul endroit tous les appels distants.

Nous disposerons donc de trois classes, nommées CatalogDelegate, CustomerDelegate et OrderDelegate, et qui respectivement délégueront les appels aux interfaces CatalogRemote, CustomerRemote et OrderRemote. Par exemple, l'extrait de code ci-après représente la classe CustomerDelegate qui délègue les appels à l'EJB CustomerBean.

Extrait de la classe CustomerDelegate

```

public final class CustomerDelegate {
    (...)
    public static Customer findCustomer(Long customerId) {
        return getCustomerRemote().findCustomer(customerId);
    }
}

```

◀ Cette méthode statique appelle la méthode findCustomer de l'EJB CustomerBean.

Celle-ci délègue l'appel à `deleteCustomer`. ▶

Cette méthode privée appelle le Service Locator pour obtenir l'interface distante. Notez le nom JNDI de l'EJB. Celui-ci est défini dans l'annotation `@Stateless` de `CustomerBean`. ▶

APPROFONDIR Business delegate

Core J2EE Patterns - Business Delegate

▶ <http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html>

Business Delegate

▶ <http://c2.com/cgi/wiki?BusinessDelegate>

UML Diagramme de séquences

Le diagramme de séquences permet de représenter des collaborations entre objets selon un point de vue temporel. On y met l'accent sur la chronologie des envois de messages. Les diagrammes de classes montrent la composition statique des classes, alors que le diagramme de séquences se penche sur la dynamique de leurs collaborations.

RAPPEL Interface locale des EJB

L'application graphique ne peut pas utiliser l'interface locale de l'EJB car elle se trouve en dehors du conteneur. Elle ne peut utiliser que la Remote.

```
public static void deleteCustomer(Customer customer) {
    getCustomerRemote().deleteCustomer(customer);
}

private static CustomerRemote getCustomerRemote() {
    CustomerRemote customerRemote;
    customerRemote = (CustomerRemote)
        ServiceLocator.getInstance().getRemoteInterface(
            "ejb/stateless/Customer");
    return customerRemote;
}
```

Vous retrouverez ci-après l'extrait de code initial permettant d'afficher une catégorie, en utilisant dorénavant la classe `CatalogDelegate`.

Écran d'affichage de la catégorie avec Business Delegate

```
public class CategoryCrudFrame extends JInternalFrame {
    (...)
    public void findActionPerformed(EventObject evt) {
        Category category= CatalogDelegate.findCategory(identifiant);
        model.setIdentifiant(category.getId());
        model.setName(category.getName());
        model.setDescription(category.getDescription());
    }
}
```

Appel d'un EJB stateless dans cette architecture

Voyons maintenant comment tout cela s'imbrique à l'aide d'un diagramme de séquences. Dans ce genre de diagramme, les objets communiquent en invoquant des opérations sur d'autres objets. On peut donc suivre visuellement les différentes interactions et les traitements réalisés par chaque objet. Prenons pour exemple la recherche d'un client et l'affichage de ses informations. Le diagramme de séquences (figure 6-2) nous montre comment l'écran Swing invoque l'EJB au travers de la classe `CustomerDelegate`.

L'application Swing appelle la méthode `findCustomer` ❶ de la classe `CustomerDelegate`. Celle-ci doit retrouver l'interface distante de l'EJB grâce au singleton `ServiceLocator`. Elle appelle donc la méthode `getInstance` ❸ puis `getRemoteInterface` ❹ en passant le nom JNDI de l'EJB (c'est-à-dire `ejb/stateless/Customer`). La classe `CustomerDelegate` utilise l'interface `CustomerRemote` et appelle la méthode `findCustomer` ❷ ce qui a pour effet d'appeler la classe d'implémentation `CustomerBean` ❸. C'est cette dernière qui manipulera l'entity manager ❹ pour obtenir l'entity `Customer`. L'application Swing se retrouve donc avec un objet `Customer` sur lequel elle peut invoquer les getters (❺ et ❻) pour afficher les informations à l'écran (figure 6-3).

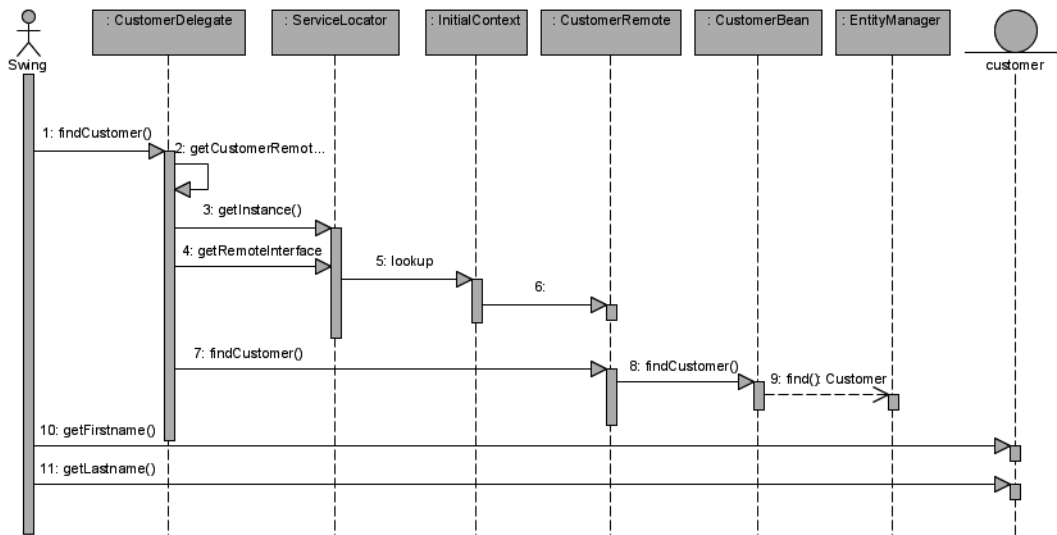


Figure 6–2 Diagramme de séquences pour afficher les données d'un client

The screenshot shows a window titled "Customer - 4 (Read only)". The window contains a form with the following fields and values:

Customer - 4 (Read only)	
Identifier	4
First Name	Antonio
Last Name	Goncalves
Date of birth	29/05/1971
Age	35
Telephone	487 487 21
Email	antonio.goncalves@gmail.com
Identification	
Login	antoncal
Password	*****
Address	
Street 1	42 Glass Street
Street 2	
City	Oakland
State	LA
Zipcode	8765
Country	USA

At the bottom of the window, there are six buttons: Find, Create, Update, Delete, Reset, and Close.

Figure 6–3
Écran d'affichage des données d'un client

REMARQUE **Look & Feel Swing**

Swing peut prendre différents aspects graphiques : Windows, Metal ou Motif dans notre exemple, mais il en existe bien d'autres selon la plate-forme et les licences.
► <http://www.javatoo.com/>

Comme nous l'avons vu précédemment, les entités peuvent se détacher de l'entity manager et devenir de simples Pojo sans persistance. C'est ce qui se passe lorsque l'écran Swing manipule l'objet Customer pour en afficher les données. Par contre, pour pouvoir traverser le réseau, cette classe doit implémenter l'interface `Serializable`.

L'avantage de ce découpage en couche réside dans le fait que l'application cliente ne connaît pas la logique métier qui se cache derrière la recherche d'un client. Elle passe juste un identifiant à une méthode qui lui retourne l'objet initialisé avec les données de la base. Les patterns Business Delegate et Service Locator nous aident d'autant mieux à faire ce découpage qu'ils isolent les responsabilités de chaque classe.

L'application graphique YAPS Pet Store

L'interface graphique des employés de la société YAPS se compose d'un menu principal (figure 6–4) qui permet d'accéder à différents sous-menus (tableau 6–1). Chacun d'eux correspond à une des fonctionnalités présentées dans les cas d'utilisation.

Figure 6–4
Menu principal

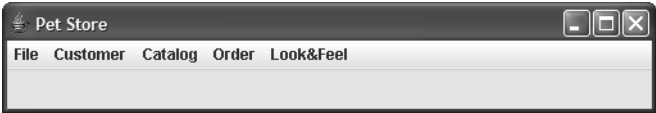


Tableau 6–1 Sous-menus de l'application

Menu	Sous-menu	Description
File	Exit	Permet à l'employé de fermer l'application
Customer	List customers	Affiche la totalité des clients de la base de données
	Manage customer	Effectue les opérations CRUD sur un client
Catalog	List categories	Affiche la totalité des catégories du catalogue
	List products	Affiche la totalité des produits du catalogue
	List items	Affiche la totalité des articles du catalogue
	Manage category	Effectue les opérations CRUD sur une catégorie
	Manage product	Effectue les opérations CRUD sur un produit
	Manage item	Effectue les opérations CRUD sur un article
Order	List orders	Affiche les bons de commande
	Manage order	Affiche l'écran permettant à l'employé de retrouver et de supprimer une commande
	Watch orders	Affiche en temps réel les commandes contenant des reptiles
Look&Feel	Metal	Change l'aspect de l'application en Metal
	Motif	Change l'aspect de l'application en Motif
	Windows	Change l'aspect de l'application en Windows

Ces menus affichent principalement deux types d'écrans :

- les listes permettant aux employés de consulter la totalité des éléments du système ;
- les écrans de création, de mise à jour, de recherche et de suppression ne manipulant qu'un seul élément à la fois.

La gestion des clients

Les employés peuvent consulter la liste des clients en cliquant sur le menu `List customers`. Cette liste comporte un bandeau de boutons permettant une action sur le client sélectionné. Ainsi, en cliquant sur une ligne de la liste, puis sur le bouton `View`, un nouvel écran s'affiche contenant les informations du client. Appuyez sur `Create` et un écran vierge s'affichera vous demandant de saisir les coordonnées d'un nouveau client, etc. (figure 6-5).

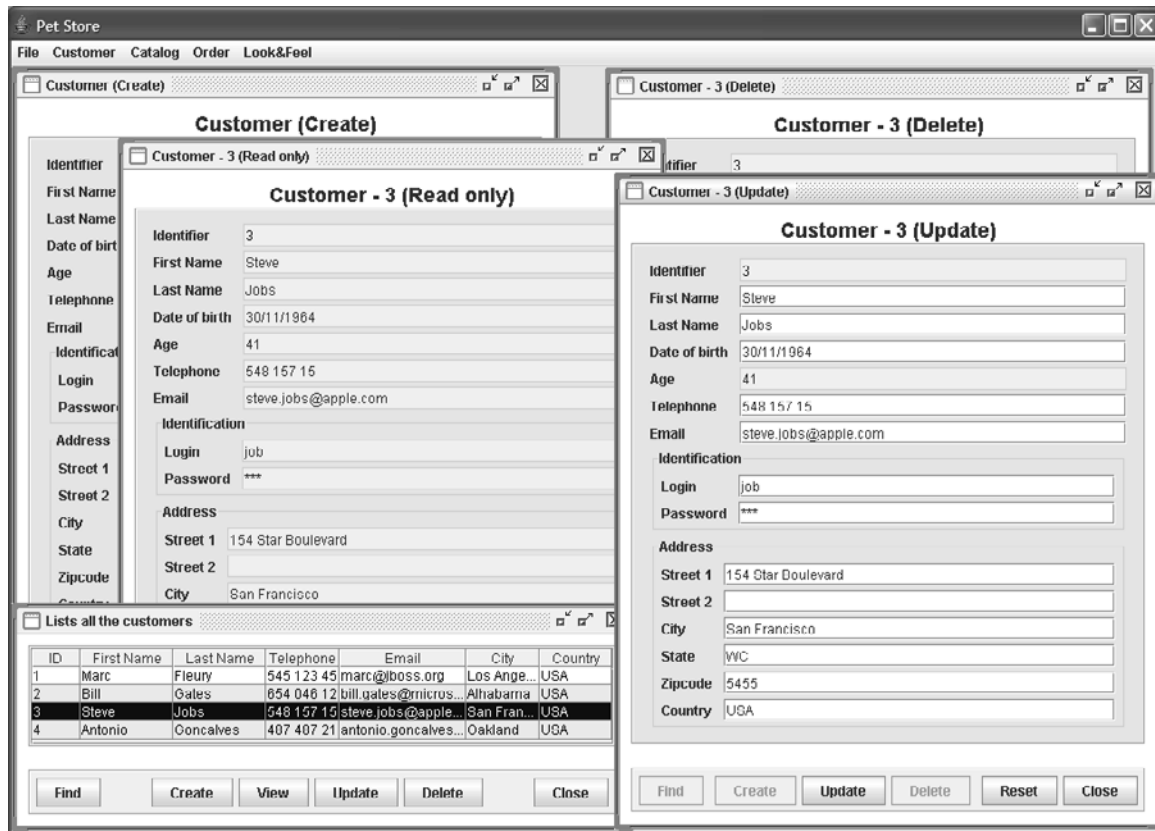


Figure 6-5 Actions possibles sur le client

Chacune de ces actions (un clic sur un bouton) appelle une méthode de la classe `CustomerDelegate` en passant les paramètres attendus. Ci-après un diagramme de classes montrant la réciprocité entre le Delegate et l'interface de l'EJB (figure 6-6).

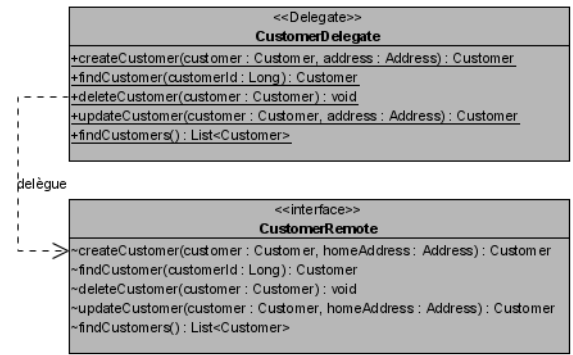


Figure 6-6
Delegate et interface distante

REMARQUE Le code des Delegate

Le code des Delegate est assez simple et répétitif. Pour des raisons de clarté, le code ainsi que les diagrammes de classes des `CatalogDelegate` et `OrderDelegate` ne seront pas détaillés.

La gestion du catalogue

Les écrans de la gestion du catalogue ressemblent à ceux du client. Une particularité, tout de même, est à noter : l'utilisation des combobox pour les relations 1:n. En effet, un produit étant rattaché à une catégorie, on retrouve une liste de catégories dans l'écran du produit. De même pour les articles et les produits (figure 6-8).

Affichage des erreurs

Lors de la création d'une catégorie, le nom ainsi que la description sont obligatoires. La validation de ces informations est faite par l'entity `Category` avant qu'il ne persiste ses données (c'est-à-dire dans une méthode annotée par `@PrePersist`).

```
@PrePersist
@PreUpdate
private void validateData() {
    if (name == null || "".equals(name))
        throw new ValidationException("Invalid name");
    if (description == null || "".equals(description))
        throw new ValidationException("Invalid description");
}
```

Si l'on essaie de créer une catégorie sans nom, l'entity lance une exception de type `ValidationException`, qui est attrapée puis affichée par le client Swing.

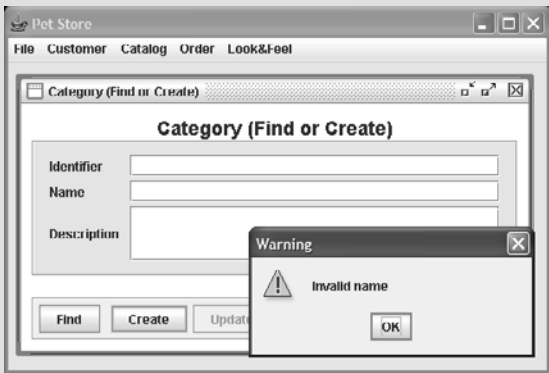


Figure 6-7 Affichage des exceptions

La gestion des bons de commande

Les bons de commande ne peuvent pas être créés ou modifiés par l'application Swing. Les employés ne peuvent qu'en obtenir la liste, en consulter le détail et éventuellement, en supprimer un (figure 6-9).

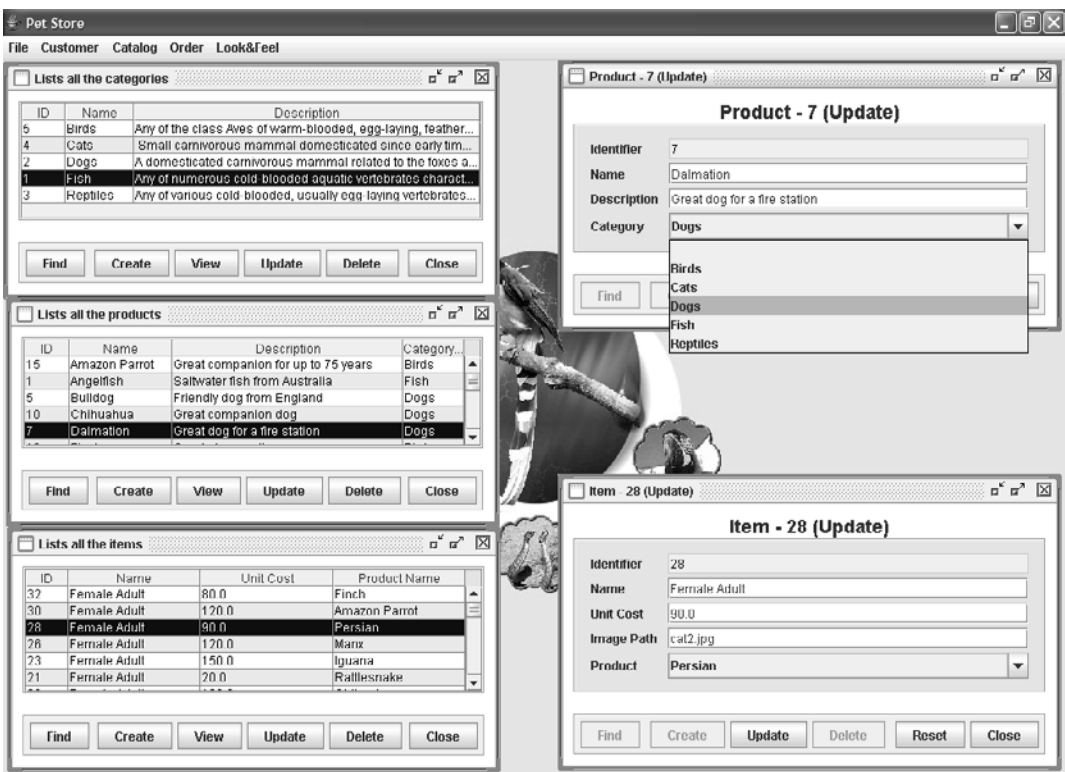


Figure 6–8
Gestion du catalogue
utilisant des combobox

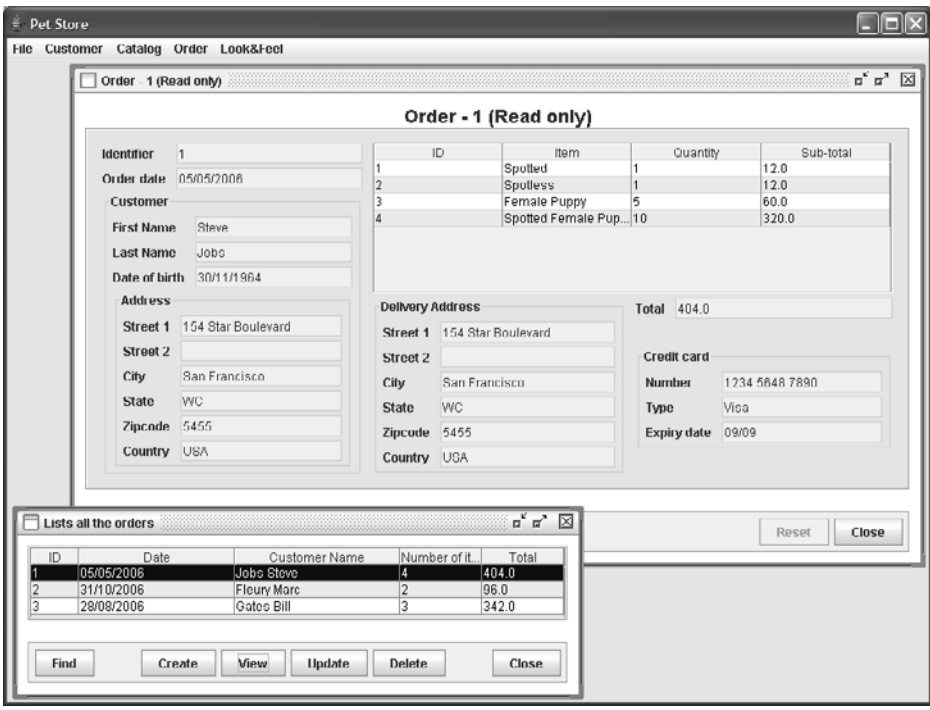


Figure 6–9
Affiche les bons de commande
et leur détail.

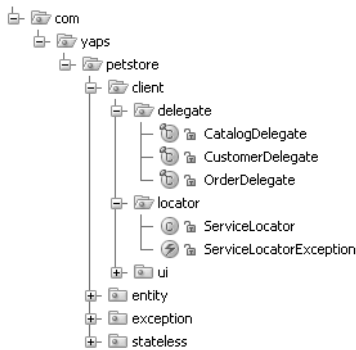


Figure 6-10 Classes Swing, Delegate et Locator

Paquetages du client Swing

Les classes de l'application graphique sont placées dans le paquetage `com.yaps.petstore.client`. Les classes Swing sont dans le sous-paquetage `ui`, les Business Delegate dans `delegate` et le Service Locator dans `locator`.

Architecture

L'architecture globale de l'application est maintenant constituée de trois couches principales : l'interface graphique, la couche de traitements et les objets persistants. Les écrans Swing utilisent les Delegate et le ServiceLocator pour accéder aux interfaces distantes des stateless beans. Ces derniers manipulent les entités à l'aide de l'entity manager.

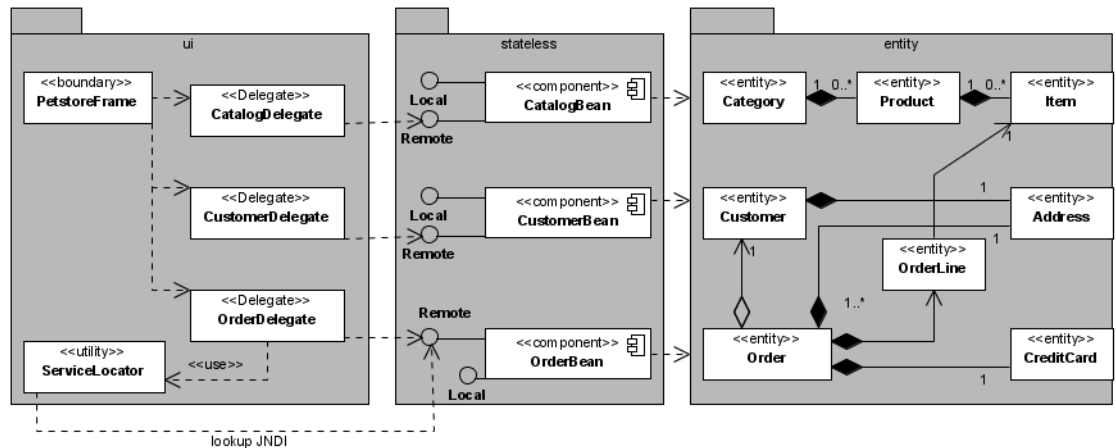


Figure 6-11 Architecture de l'application avec l'interface graphique

Arborescence des répertoires

Les classes java sont développées dans le répertoire `src`, compilées dans `classes` et packagées dans `build`.

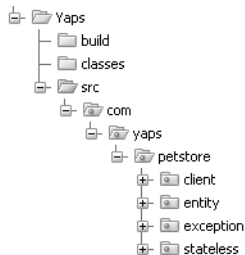


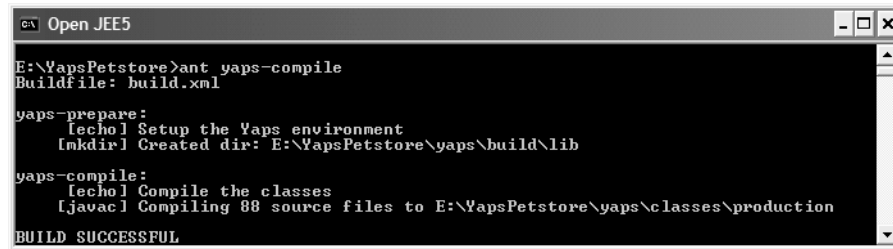
Figure 6-12 Répertoires de l'application

Exécuter l'application

Maintenant que nous avons étudié les écrans Swing et leur imbrication dans l'architecture, il ne nous reste qu'à exécuter l'application. Pour cela, il faut d'abord compiler les classes développées jusqu'ici, les packager dans des fichiers `.jar`, puis les déployer sur le serveur GlassFish. Les tâches Ant s'occuperont d'effectuer tous ces traitements.

Compiler

Avant tout, il faut compiler les classes qui se trouvent dans le répertoire `src`. La tâche Ant `yaps-compile` se charge de les compiler et de les placer dans le répertoire `classes` (figure 6–13). Si l'on souhaite supprimer tous les fichiers `.class` et les répertoires de travail (`build` et `classes`), il suffit d'utiliser la tâche Ant `yaps-clean`.



```

C:\ Open JEE5
E:\YapsPetstore>ant yaps-compile
Buildfile: build.xml

yaps-prepare:
[echo] Setup the Yaps environment
[mkdir] Created dir: E:\YapsPetstore\yaps\build\lib

yaps-compile:
[echo] Compile the classes
[javac] Compiling 88 source files to E:\YapsPetstore\yaps\classes\production
BUILD SUCCESSFUL
  
```

ANT Les tâches dans `build.xml` et `admin.xml`

Les fichiers contenant les tâches Ant (`build.xml` et `admin.xml`) sont décrits en annexe et téléchargeables sur :
 ▶ <http://www.antonigoncalves.org>

Figure 6–13
Exécution de la tâche `yaps-compile`

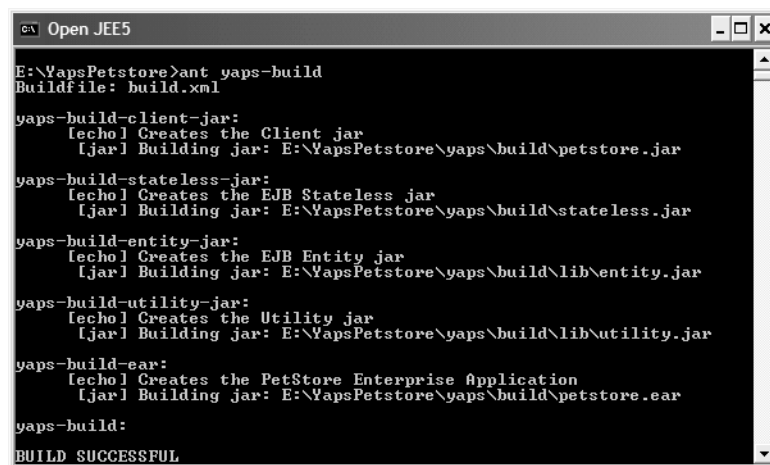
Pour pouvoir compiler, notre application requiert l'utilisation de certaines librairies du serveur GlassFish (toutes les classes et annotations de Java EE 5 importées dans notre code). La tâche Ant se charge de rajouter ces librairies externes dans le classpath. Vous retrouverez la liste de ces librairies en annexes dans le fichier `build.xml`.

La variable `classpath`

La variable `classpath` définit les répertoires où doit être recherché le byte-code et/ou les sources des classes Java lors de la compilation et de l'exécution.

Packager

Une fois les classes compilées, il est nécessaire de les packager dans des fichiers d'archive. Ces archives constituent le moyen standard d'emballer toutes les parties de l'application (bytecode Java, images, fichiers de propriétés, etc.) afin d'être exécutées ou déployées. On exécute la tâche `yaps-build` pour créer les fichiers d'archive de l'interface graphique et de l'application serveur (figure 6–14). Ces fichiers sont placés dans le répertoire `build`.



```

C:\ Open JEE5
E:\YapsPetstore>ant yaps-build
Buildfile: build.xml

yaps-build-client-jar:
[echo] Creates the Client jar
[jar] Building jar: E:\YapsPetstore\yaps\build\petstore.jar

yaps-build-stateless-jar:
[echo] Creates the EJB Stateless jar
[jar] Building jar: E:\YapsPetstore\yaps\build\stateless.jar

yaps-build-entity-jar:
[echo] Creates the EJB Entity jar
[jar] Building jar: E:\YapsPetstore\yaps\build\lib\entity.jar

yaps-build-utility-jar:
[echo] Creates the Utility jar
[jar] Building jar: E:\YapsPetstore\yaps\build\lib\utility.jar

yaps-build-ear:
[echo] Creates the PetStore Enterprise Application
[jar] Building jar: E:\YapsPetstore\yaps\build\petstore.ear

yaps-build:
BUILD SUCCESSFUL
  
```

Figure 6–14
Exécution de la tâche `yaps-build`

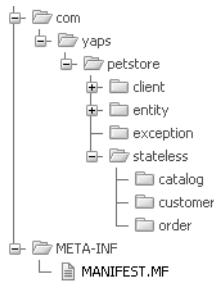


Figure 6-15 Contenu du fichier petstore.jar

Le fichier MANIFEST.MF

Dans l'arborescence graphique du fichier `petstore.jar`, vous voyez apparaître un fichier `MANIFEST.MF`. Celui-ci est indispensable aux fichiers d'archive. Présent dans le répertoire `META-INF`, il peut contenir de nombreuses informations (sous la forme clé/valeur) sur l'archive et son contenu.



Figure 6-16 Contenu du fichier petstore.ear

Les fichiers d'archive

Il existe plusieurs types de fichiers d'archive pour packager une application Java EE :

- les `.jar` (java archive) pour les classes Java et les EJB ;
- les `.war` (web archive) pour les applications web (servlet, jsp, jsf, images, html, etc.) ;
- les `.ear` (enterprise archive) pour contenir les fichiers `.jar` et `.war`.

ANNOTATIONS Les descripteurs XML

Lorsqu'on déployait des EJB en J2EE 1.4, il fallait fournir des descripteurs de déploiement XML (`ejb-jar.xml` et `application.xml`). Les annotations de Java EE 5 nous permettent de nous affranchir de ces fichiers devenus optionnels.

Interface graphique

La totalité de l'application graphique est contenue dans le fichier `petstore.jar`. Il y a, bien sûr, toutes les classes Swing du paquetage `com.yaps.petstore.client.ui`, mais aussi les Business Delegate et le Service Locator. Pour pouvoir manipuler les entités, il faut les rajouter dans ce `.jar` ainsi que les exceptions et les interfaces distantes des EJB `stateless` (c'est-à-dire `CatalogRemote`, `CustomerRemote` et `OrderRemote`). Le fichier `petstore.jar` ne requiert ni les interfaces locales (car uniquement accessibles à l'intérieur du conteneur), ni les classes d'implémentation des EJB `stateless`.

Application serveur

Côté serveur, l'application est packagée dans un fichier `.ear`. Ce type particulier d'archive est utilisé pour les applications d'entreprise (entreprise archive) et contient plusieurs autres fichiers. Sa structure est représentée graphiquement ci-après.

Les classes et interfaces des EJB session (`Remote`, `Local` et `Bean`) se trouvent dans le fichier `stateless.jar`, les entités dans `entity.jar` et les exceptions, et autres classes utilitaires, dans `utility.jar`.

Remarquez la disposition de ces fichiers à l'intérieur de l'EAR. `stateless.jar` est placé à la racine, alors que `entity.jar` et `utility.jar` sont dans le sous-répertoire `lib`. Ce répertoire est particulier puisqu'il permet de partager les classes à tout le fichier `.ear`. En effet, comme nous le verrons par la suite, les EJB `stateful` et `asynchrones` auront, eux aussi, besoin des entités et des exceptions. Pour ne pas dupliquer ces classes dans les différents fichiers `.jar`, on les rend accessible en les plaçant dans le répertoire `lib`.

Dernière particularité, le fichier `persistence.xml` se trouve dans le répertoire `META-INF` du fichier `entity.jar`. Ce fichier doit être déployé dans ce répertoire pour être pris en compte. Rappelez-vous que ce fichier permet d'informer le conteneur de l'unité de persistance qu'il doit utiliser.

Déploier

Le déploiement n'est pas nécessaire pour la partie graphique qui est considérée comme une simple application Java SE. Par contre, il est indispensable pour la partie serveur puisque les EJB doivent s'exécuter à l'intérieur d'un conteneur.

Avant toute chose, le serveur GlassFish et la base de données Derby doivent être démarrés. Pour cela, utilisez les tâches Ant d'administration :

```
%PETSTORE_HOME%\> ant -f admin.xml start-domain
%PETSTORE_HOME%\> ant -f admin.xml start-db
```

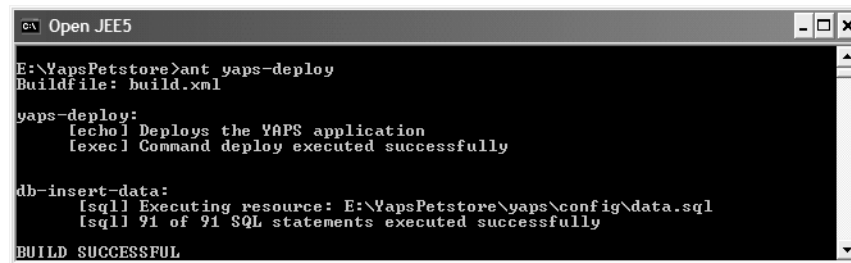
Pour vérifier que GlassFish fonctionne, rendez-vous à l'adresse `http://localhost:8080`. La page d'invite s'affiche. Vous pouvez aussi consulter les logs et vérifier que le message suivant apparaît :

```
Application server startup complete.
```

GLASSFISH Consulter les logs

Pour lire les logs du serveur GlassFish vous pouvez, soit consulter le fichier `%GLASSFISH_HOME%\domains\petstore\logs`, soit vous connecter à la console d'administration. Pour cela, allez à l'adresse `http://localhost:8282` puis saisissez le nom de l'utilisateur `admin` et son mot de passe `adminpwd`. Cliquez sur le menu *Application Server > View Log Files*.

Ce n'est qu'une fois le serveur démarré que l'on peut déployer l'application. Pour cela, utilisez la tâche Ant `yaps-deploy` (figure 6-17). Celle-ci fait appel au programme d'administration de GlassFish (`asadmin`) et lui passe en paramètre le fichier `petstore.ear`.



```
C:\ Open JEE5
E:\YapsPetstore>ant yaps-deploy
Buildfile: build.xml

yaps-deploy:
[echo] Deploys the YAPS application
[exec] Command deploy executed successfully

db-insert-data:
[sql] Executing resource: E:\YapsPetstore\yaps\config\data.sql
[sql] 91 of 91 SQL statements executed successfully

BUILD SUCCESSFUL
```

Figure 6-17 Exécution de la tâche `yaps-deploy`

Le déploiement permet au serveur d'applications de lire le fichier `.ear` et d'en extraire les informations utiles. Ce processus va donc créer la base de données à partir des annotations JPA, découvrir les EJB grâce aux annotations `@javax.ejb.Stateless`, accéder au service de nommage pour y stocker les interfaces distantes, etc. Chacune de ces étapes affichera des traces dans GlassFish.

TopLink va donc créer les différentes tables et colonnes à partir des entités et tracera ces actions dans son fichier de logs.

Java Web Start

Nous aurions pu déployer le client Swing avec Java Web Start. Cette technologie, développée avec la plate-forme Java 2 et incluse dans le JRE depuis la version 1.4, permet le déploiement d'applications Java SE à travers le réseau. On peut ainsi installer une application grâce à un simple clic dans un navigateur. Java Web Start assure alors la mise à jour automatique des nouvelles versions de l'application et utilise un cache local pour accélérer sa réutilisation ultérieure.

► <http://java.sun.com/products/javawebstart/>

ANT Deploy, undeploy

Une fois l'application déployée à l'aide de la tâche `yaps-deploy`, on peut la supprimer du serveur GlassFish en utilisant la tâche `yaps-undeploy`.

Création automatique des tables

Le schéma de la base de données peut être automatiquement créé lors du déploiement des entités. Cette information est contenue dans le fichier `persistence.xml` (packagé dans `entity.jar`) et peut être modifiée.

```
<persistence>
  <persistence-unit name="petstorePU">
    <jta-data-source>jdbc/petstoreDS</jta-data-source>
    <properties>
      <property name="toplink.target-database"
        value="Derby"/>
      <property name="toplink.ddl-generation"
        value="drop-and-create-tables"/> ❶
      <property name="toplink.create-ddl-jdbc-file-name"
        value="create.sql"/> ❷
      <property name="toplink.drop-ddl-jdbc-file-name"
        value="drop.sql"/> ❸
      <property name="toplink.logging.level"
        value="FINEST"/> ❹
    </properties>
  </persistence-unit>
</persistence>
```

L'attribut `toplink.ddl-generation` ❶ permet à TopLink de savoir quelle action entreprendre lors du déploiement. Il accepte les valeurs suivantes :

- `drop-and-create-tables` : lors du déploiement, le schéma de la base de données est supprimé puis recréé.
- `create-tables` : à chaque déploiement, TopLink exécute les requêtes de création des tables, même si elles existent déjà. Des avertissement peuvent alors apparaître.
- `none` : TopLink n'entreprend aucune action, à vous de garantir l'existence des tables.

Les autres propriétés de ce fichier signifient que les scripts SQL de création et de suppression générés par TopLink se trouvent dans les fichiers `create.sql` ❷ et `drop.sql` ❸. Pour augmenter ou diminuer le niveau de traces, on peut modifier le paramètre `toplink.logging.level` ❹.

Notez que tous ces paramètres sont propres à TopLink qui est le moteur de persistance de GlassFish. Une autre implémentation de JPA, comme Hibernate par exemple, utilisera d'autres propriétés.

► <http://www.oracle.com/technology/products/ias/toplink/jpa/resources/toplink-jpa-extensions.html#Java2DBSchemaGen>

Traces de création du schéma de la base de données

The alias name for the entity class [class com.yaps.petstore.entity.catalog.**Product**] is being defaulted to: Product.

The column name for element [private java.lang.Long com.yaps.petstore.entity.catalog.Product.**id**] is being defaulted to: ID.

The column name for element [private java.lang.String com.yaps.petstore.entity.catalog.Product.**name**] is being defaulted to: NAME.

The column name for element [private java.lang.String com.yaps.petstore.entity.catalog.Product.**description**] is being defaulted to: DESCRIPTION.

Notez que suite au déploiement, la tâche `Ant yaps-deploy` en profite pour insérer des données dans la base via la tâche `db-insert-data` (qui se trouve dans le fichier `build.xml`). Celle-ci exécute le fichier `data.sql` contenant les ordres SQL `insert` et permet ainsi d'avoir un jeu de données initial pour utiliser l'application (voir annexe B).

En ce qui concerne nos trois `stateless session beans`, ils sont enregistrés dans JNDI et deviennent accessibles de manière distante :

Traces d'un enregistrement des interfaces dans JNDI fait au déploiement

```
java:comp/env/
com.yaps.petstore.stateless.customer.CustomerBean/
em;|naming.bind
RemoteBusinessJndiName: ejb/stateless/Customer; remoteBusIntf:
com.yaps.petstore.stateless.customer.CustomerRemote

java:comp/env/com.yaps.petstore.stateless.catalog.CatalogBean/
em;|naming.bind
RemoteBusinessJndiName: ejb/stateless/Catalog; remoteBusIntf:
com.yaps.petstore.stateless.catalog.CatalogRemote

java:comp/env/com.yaps.petstore.stateless.order.OrderBean/
em;|naming.bind
RemoteBusinessJndiName: ejb/stateless/Order; remoteBusIntf:
com.yaps.petstore.stateless.order.OrderRemote
```

Une fois le fichier `petstore.ear` déployé et toutes ces étapes passées avec succès, vous pouvez vous rendre sur la console d'administration de GlassFish pour consulter ces informations (<http://localhost:8282>). Vous trouverez, entre autres, le contenu de l'arbre JNDI avec tous les stateless beans déployés (figure 6–18).

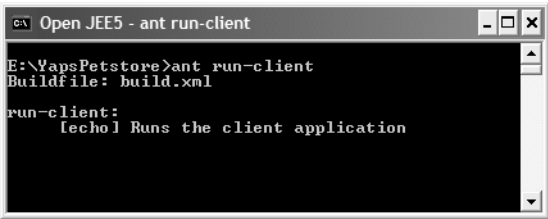


Figure 6–18
Stateless beans dans JNDI

Exécuter

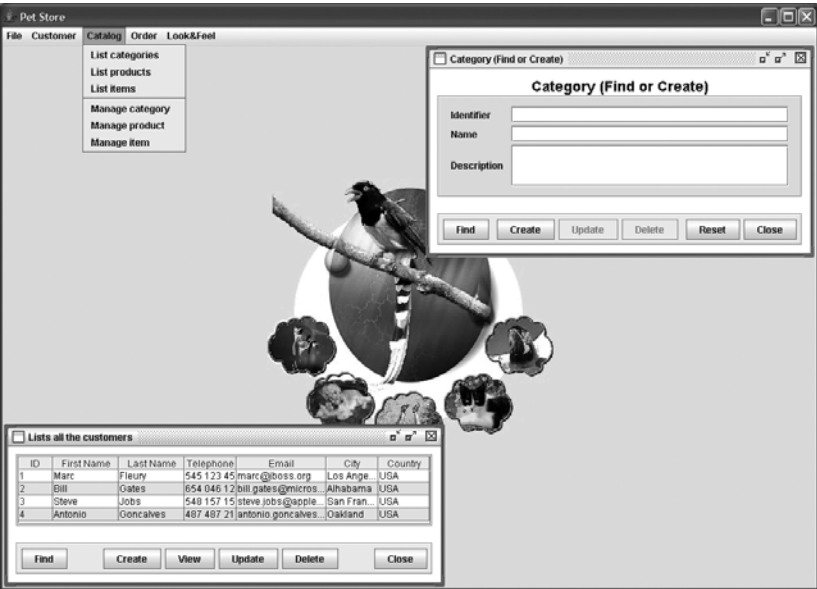
Toutes nos classes sont compilées, packagées et déployées sur le serveur d'applications. La base de données est créée et contient des données. Il ne reste plus qu'à exécuter l'interface graphique pour effectuer les traitements métier demandés par les employés de la société YAPS. Pour ce faire, utilisez la tâche `Ant run-client` (figure 6–19).

Figure 6–19
Exécution de la tâche run-client



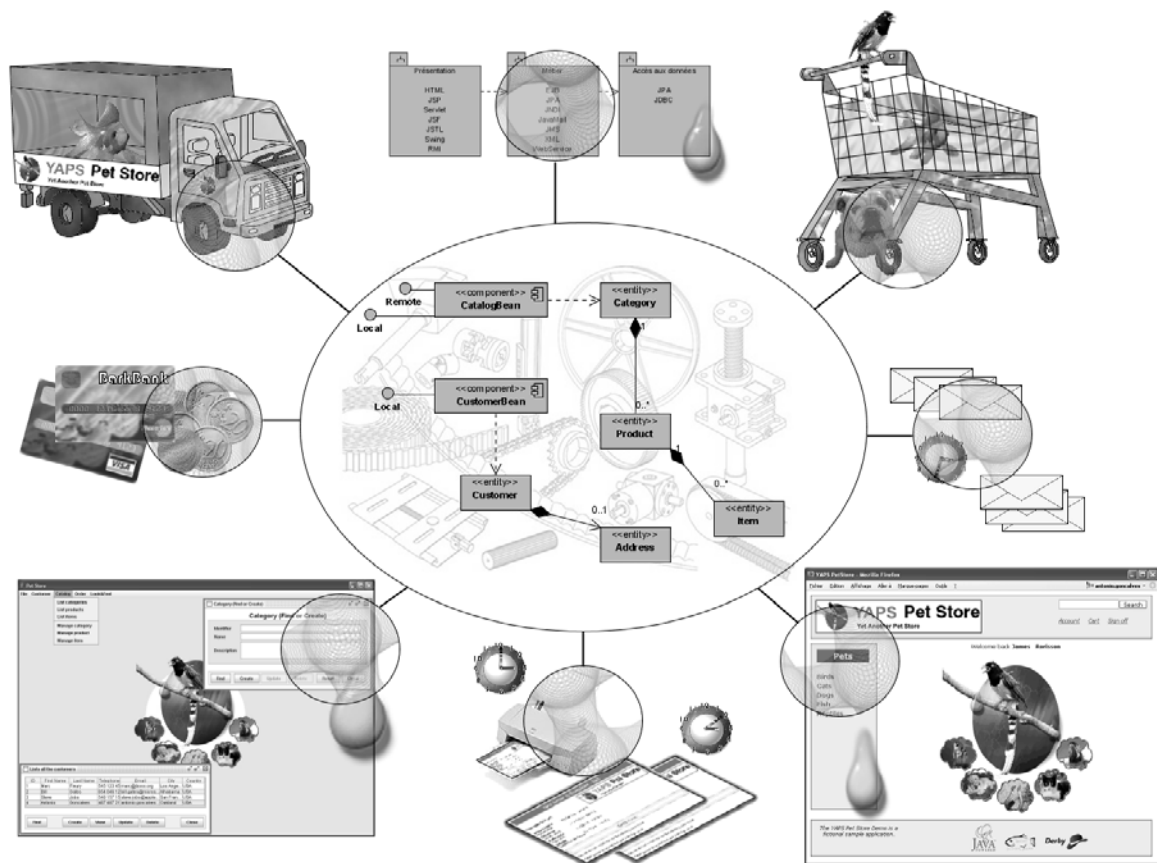
Cette tâche lance la classe du menu principal (PetstoreFrame) qui se trouve dans le fichier d'archive petstore.jar. Pour pouvoir s'exécuter convenablement, certaines librairies GlassFish sont rajoutées au class-path. Il ne vous reste plus maintenant qu'à utiliser l'application Swing (figure 6–20) avant de passer à l'interface web.

Figure 6–20
Menu principal de l'application cliente



En résumé

Ce chapitre nous a montré comment compiler, packager et déployer l'application YAPS Pet Store dans le serveur GlassFish. Une interface graphique Swing a également été développée pour permettre aux employés de gérer le catalogue, les clients et les bons de commande du système. Il nous faut maintenant développer une interface web pour permettre aux internautes de consulter le catalogue d'articles et de se créer un compte. Ces deux IHM délèguent les traitements métier à la couche de stateless session beans que nous avons vu dans le précédent chapitre.



7

chapitre



Interface web

L'application est à présent déployée et utilisable par un client Swing. Il nous faut maintenant développer l'interface web qui sera utilisée par les internautes et clients de la société YAPS. Ce chapitre introduit les technologies servlet, JSP et JSTL avant de présenter JSF et son modèle MVC. L'application web utilise JSF et dialogue avec la couche de stateless beans.

SOMMAIRE

- Interface web de l'application
- Visualiser le catalogue
- Rechercher les articles
- Gérer le compte client
- Le duo servlet-JSP
- JSF et le modèle MVC
- Appel local des stateless bean
- L'injection
- Navigation entre pages

MOTS-CLÉS

- Servlet
- JSP
- JSTL
- JSF
- Managed bean
- EL et UEL
- MVC
- IoC

APPROFONDIR HTML & CSS

- ▶ <http://www.w3.org/MarkUp/>
- ▶ <http://www.htmlprimer.com/>
- ▶ <http://www.w3.org/Style/CSS/>
- ▶ <http://www.w3schools.com/css/>

📖 Éric Sarrion, *Introduction à HTML et CSS*, O'Reilly, 2006

/// CGI, ASP et PHP

CGI (*Common Gateway Interface*), inventé en 1993, permet d'exécuter un programme sur un serveur et de renvoyer le résultat à un navigateur Internet.

PHP (*Hypertext Preprocessor*) est un langage de script HTML inspiré des langages C, Java et Perl. ASP (*Active Server Page*) est un langage propriétaire de Microsoft pour les développements web.

/// HTTP

HyperText Transfer Protocol est un protocole de communication pour transférer les documents (HTML, image, etc.) entre le serveur HTTP et le navigateur web. Ce protocole utilise plusieurs types de requêtes (commandes) pour demander au serveur d'effectuer une action :

- GET : demander une ressource au serveur ;
 - HEAD : ne demander que des informations sur la ressource ;
 - POST : ajouter une nouvelle ressource ;
 - PUT : remplacer ou ajouter une ressource sur le serveur ;
 - DELETE : supprimer une ressource du serveur.
- ▶ <http://www.w3.org/Protocols/>
-

APPROFONDIR Servlet

Java Servlet Technology

- ▶ <http://java.sun.com/products/servlet/>

📖 Jean-Luc Délage, *JSP et Servlets efficaces : production de sites dynamiques en Java*, Dunod, 2007

📖 Jason Hunter, William Crawford, *Servlets Java : Guide du programmeur*, O'Reilly, 2002¹

Les employés ont leur application, il est temps maintenant de développer la partie web pour les internautes et les clients. Elle leur permettra de consulter le catalogue, de rechercher des articles par mots-clés mais aussi de se créer un compte et de devenir client (voir cas d'utilisation « Visualiser les articles du catalogue », « Rechercher un article », « Se créer un compte », « Se connecter et se déconnecter », « Consulter et modifier son compte »).

Le duo Servlet-JSP

Une application web, c'est avant tout un aspect visuel. Les pages qui constituent le site, utilisent plusieurs artefacts pour avoir un rendu graphique. Tout d'abord, une page est essentiellement constituée de balises HTML qui sont interprétées et affichées par un navigateur. Conjointement à HTML, on peut aussi utiliser des feuilles de style (*Cascading Style Sheets* ou CSS) pour enrichir les possibilités graphiques.

Cependant, un site web n'est pas uniquement constitué de pages statiques, de sons, d'images et de couleurs. Il est nécessaire d'y ajouter des données provenant de traitements côté serveur pour obtenir un contenu dynamique. Pour cela, la plate-forme Java EE met à disposition plusieurs spécifications comme les servlets, les pages JSP, les taglibs ainsi que JSF.

Les servlets

Bien que le développement de l'application YAPS Pet Store n'utilise pas les servlets mais les JSP (*Java Server Pages*), les balises JSTL (*JSP Standard Tag Library*) et JSF (*Java Server Faces*), une petite introduction est nécessaire.

Fonctionnant côté serveur au même titre que CGI, ASP ou PHP, les servlets permettent de gérer des requêtes HTTP, d'effectuer des traitements, d'appeler des EJB, des services, et de fournir au navigateur une réponse sous forme de page web. Les servlets implémentent les classes et les interfaces des paquetages `javax.servlet` (classes génériques indépendantes du protocole) et `javax.servlet.http` (spécifique au protocole HTTP). Elles étendent la classe `javax.servlet.GenericServlet` ou, plus couramment, `javax.servlet.http.HttpServlet`.

En pratique, les servlets, socle omniprésent de Java EE, ne sont plus utilisées directement dans la programmation d'un site web. En effet, elles souffrent de nombreuses lacunes, comme l'absence de séparation entre les traitements et la présentation ou encore l'aiguillage d'une servlet à une autre.

Le code ci-après nous montre un extrait de servlet affichant la liste des produits pour une catégorie donnée.

Extrait de servlet mêlant traitement et présentation

<pre> public class ShowProductsServlet extends HttpServlet { public void service(HttpServletRequest request, HttpServletResponse response) throws ServletException { response.setContentType("text/html"); PrintWriter out = response.getWriter(); Context initialContext = new InitialContext(); CatalogLocal catalogLocal = (CatalogLocal) initialContext.lookup("ejb/stateless/Catalog"); Long categoryId = Long.valueOf(request.getAttribute("categoryId")); Category category = catalogLocal.findCategory(categoryId); List<Product> products = category.getProducts(); out.println("<HTML>"); out.println("<HEAD>"); out.println("<TITLE>Display Products</TITLE>"); out.println("</HEAD>"); out.println("<BODY>"); out.println("<TABLE>"); for (Product product : products) { out.println("<TR>"); out.println("<TD>"); out.println(""); out.println(product.getName()); out.println(""); out.println("
"); out.println(product.getDescription()); out.println("</TD>"); out.println("</TR>"); } out.println("</TABLE>"); out.println("</BODY>"); out.println("</HTML>"); out.close(); } } </pre>	◀ Une servlet étend la classe <code>HttpServlet</code>. ◀ Point d'entrée d'une servlet. ◀ Lookup JNDI pour obtenir l'interface locale de l'EJB <code>CatalogBean</code>. ◀ L'EJB retourne la liste des produits pour une catégorie donnée. ◀ La servlet prépare la réponse à renvoyer au navigateur sous forme de HTML. ◀ On itère la liste des produits pour en afficher le nom et la description sous forme de tableau (balises <code>TABLE</code>, <code>TR</code> et <code>TD</code>).
---	---

Comme le montre cet exemple, le code n'est pas très élégant et peut rapidement devenir illisible. On y mélange des traitements Java (l'appel à l'EJB) et de la présentation HTML (souvent avec des images, des styles, des animations, etc.). Dans le cas de pages complexes, le code devient vite difficile à maintenir. Pour ne pas se retrouver dans ce cas de figure, les JSP sont venues simplifier le développement de pages web.

APPROFONDIR JSP

- 📖 Anne Tasso, Sébastien Ermacore, *Initiation à JSP*, Eyrolles, 2004
 - 📖 Simon Brown, Sam Dalton, Daniel Jepp, Dave Johnson, Sing Li, Matt Raible, *Pro JSP 2*, Apress, 2005
 - 📖 Jean-Luc Déléage, *JSP et Servlets efficaces*, Dunod, 2007
 - 📖 François-Xavier Sennesal, *JSP avec Eclipse et Tomcat*, ENI, 2007
- JavaServer Pages Technology :*
 ▶ <http://java.sun.com/products/jsp/>
The JSP Resource Index :
 ▶ <http://www.jspin.com/>

Look up JNDI pour obtenir l'interface locale du stateless CatalogBean.

L'EJB retourne la liste des produits pour une catégorie donnée.

On itère la liste des produits.

On affiche le nom et la description du produit.

WEB Extensions des pages JSP

Les fichiers de pages JSP portent l'extension `.jsp`. Lorsqu'une page est développée pour produire de l'XHTML, elle a l'extension `.jspx`. Pour les fragments de page (en-tête, menu, pied de page), il est courant d'utiliser l'extension `.jspf`.

Les JSP

Les JSP permettent l'affichage de contenus dynamiques de manière plus lisible que les servlets. Elles consistent en une page HTML, incluant des directives JSP et du code Java, page qui sera ensuite précompilée en servlet, puis exécutée dans un conteneur web. La page est principalement constituée de balises HTML pour l'affichage, et de code Java pour les traitements. C'est un peu l'inverse de la servlet.

L'exemple ci-après nous montre une JSP qui effectue le même traitement, c'est-à-dire afficher la liste des produits pour une catégorie.

Extrait de JSP appelant un EJB pour afficher des produits

```
<%@ page import="com.yaps.petstore.entity.catalog.Product" %>
<%@ page import="java.util.List" %> ❶
<%@ page import="javax.naming.InitialContext" %>
(...)
<TABLE> ❷

  <% ❸
    Context initialContext = new InitialContext();
    CatalogLocal catalogLocal = (CatalogLocal) ❹
      initialContext.lookup("ejb/stateless/Catalog");

    Long categoryId = Long.valueOf(
      request.getAttribute("categoryId"));
    Category category = catalogLocal.findCategory(categoryId);
    List<Product> products = category.getProducts(); ❺

    for (Product product : products) { ❻
  %> ❸
    <TR>
      <TD>
        <A href=""><%= product.getName() %></A> ❷
        <BR><%= product.getDescription()%>

      </TD>
    </TR>
  <% } %> ❻
</TABLE> ❷
```

La directive import ❶ possède le même comportement que l'import en Java. Dans notre cas, elle permet d'importer l'entity Product, la classe `java.util.List` et d'autres classes comme celles de JNDI. Grâce aux scriptlets ❸, on peut inclure du code Java dans la page et, par exemple, rechercher un EJB dans JNDI ❹, puis l'invoquer pour qu'il retourne la liste des produits ❺. À l'aide d'une boucle for ❻, on affiche dans un tableau HTML ❷ le nom du produit ainsi que sa description ❷ en utilisant l'expression (`<%=`).

Les balises JSP

Pour déclarer, exécuter ou manipuler des objets Java, les JSP utilisent plusieurs types de balises :

- La directive (`<%@ %>`) est une instruction insérée dans des balises HTML spécifiques.
`<%@ page import="java.util.Date"%>`
- La déclaration (`<%! %>`) permet d'insérer du code déclaratif dans la JSP. Elle peut être utilisée pour définir une variable globale à la classe ou pour créer des méthodes Java.
`<%! int i = 0; %>`
- Le scriptlet (`<% %>`) est utilisé pour placer du code dans la JSP. C'est généralement l'élément utilisé pour placer tout code Java, sauf les méthodes et les variables de classe.
`<% int i = 0; out.println("Affiche la valeur d'une variable : " + i); %>`
- L'expression (`<%= %>`) sert à afficher du texte ou un résultat. Ce code est équivalent à un appel `out.print()`.
Valeur de variable : `<%= i %>`
- L'action permet de fournir des instructions à l'interpréteur JSP.
`<jsp:useBean id="address" class="com.yaps.petstore.entity.Address"/>`

Vous l'aurez compris, ce genre de page mêlant code Java et balises HTML est à peine plus lisible que la servlet que nous avons vue précédemment. La page peut vite devenir compliquée à lire, à maintenir, et rapidement compter plusieurs centaines de lignes de code.

Pour résoudre ce problème, il est nécessaire de séparer les responsabilités entre composants : les servlets exécutent les traitements, appellent les EJB, gèrent les exceptions et renvoient le résultat à la JSP qui l'affiche. Cette façon de décorrélérer traitement (servlet) et présentation (JSP) est inspirée du design pattern MVC (*Model-View-Controller*).

Le design pattern MVC

Le design pattern Modèle-Vue-Contrôleur permet d'organiser une application en trois composants principaux :

- un modèle, qui correspond aux données de l'application (dans notre cas, les entités) ;
- une vue, qui correspond à la présentation visuelle de l'application (la JSP) ;
- un contrôleur, qui définit l'état de la vue en fonction des données gérées par le modèle (la servlet).

APPROFONDIR MVC

Model View Controller :

- <http://c2.com/cgi/wiki?ModelViewController>

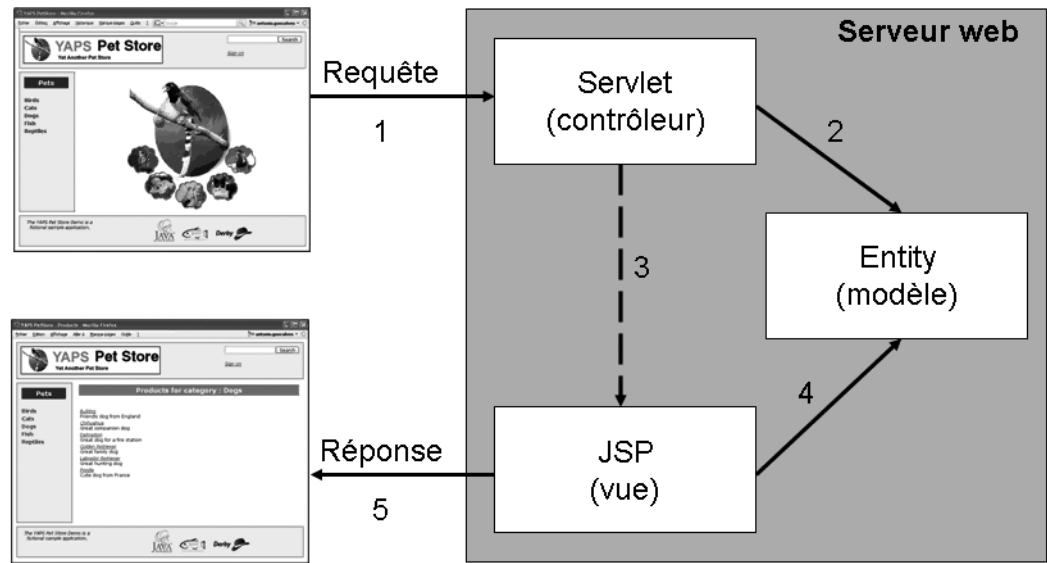


Figure 7-1
Le design pattern MVC

APPROFONDIR **Requête et réponse HTTP**

L'objet `javax.servlet.ServletException` encapsule la requête du client, c'est-à-dire qu'il contient l'ensemble des paramètres passés à la servlet (informations sur l'environnement du client, cookies du client, URL demandée, etc.). L'objet `ServletResponse`, quant à lui, permet de renvoyer une réponse au navigateur. Il est ainsi possible de créer des en-têtes HTTP (headers), d'envoyer des cookies au navigateur du client, etc.

- 1 Par le biais d'une page web, l'utilisateur émet une requête HTTP au serveur web en cliquant sur un lien ou sur un bouton. Cette requête est prise en charge par le contrôleur (servlet).
- 2 Le contrôleur exécute les traitements nécessaires (appelle un EJB stateless par exemple) et récupère le modèle, c'est-à-dire les entités.
- 3 Le contrôleur sélectionne alors la JSP qui sera en charge de la construction de la réponse et lui transmet les entités contenant les données à afficher.
- 4 La JSP construit la réponse en faisant appel aux accesseurs des entités.
- 5 La réponse HTTP est transmise au navigateur qui l'affiche sous forme de page web.

Pour reprendre l'exemple de l'affichage des produits, imaginez que l'appel à l'EJB se fasse dans une servlet et que celle-ci transmette la liste des produits à une JSP. Voici ce à quoi ressemblerait cette JSP.

Exemple de JSP recevant des données d'une servlet

```
<%@ page import="com.yaps.petstore.entity.catalog.Product" %>
<%@ page import="java.util.List" %>
<TABLE>

<jsp:useBean id="products" class="List<Product>" 1
scope="request"/>
```

La servlet transmet la liste des produits à la JSP. La portée de cette liste est la requête.

```

<%
  for (Product product : products) { ❷
%>
  <TR>
    <TD>
      <A href=""><%= product.getName() %></A> ❸
      <BR><%= product.getDescription() %>
    </TD>
  </TR>
<% } %> ❹
</TABLE>

```

◀ On parcourt la liste des produits.

◀ On affiche le nom et la description du produit dans un tableau HTML.

La servlet invoque un EJB, reçoit la liste des produits, et passe cette liste à la JSP ❶ qui peut alors y accéder grâce à la directive `jsp:useBean`. On continue à utiliser du code Java dans les scriptlets pour parcourir ❷ la liste des produits. On affiche alors le nom du produit et sa description ❸ dans un tableau HTML.

Comme on peut le constater en comparant les trois extraits de code précédents (servlet mêlant traitement et présentation, JSP appelant un EJB et JSP recevant les données d'une servlet), le design pattern MVC apporte une nette amélioration dans la lisibilité et la maintenance du code tout en séparant les responsabilités. La JSP n'est plus encombrée de code JNDI pour appeler l'EJB par exemple. Il reste tout de même encore un peu de code Java dans la page (la boucle `for` dans notre exemple) qui complique la lecture du code (la fin de la boucle `for` qui se trouve en bas de la page ❹ n'est pas facilement repérable). Les bibliothèques standards de balises, ou JSTL, peuvent encore améliorer la compréhension de la page.

ARCHITECTURE **Separation of Concerns (SoC)**

La séparation des responsabilités permet de séparer différents aspects d'un problème afin de pouvoir se concentrer plus efficacement sur chacun.

- ▶ http://en.wikipedia.org/wiki/Separation_of_concerns
- ▶ <http://www.latece.uqam.ca/publications/mili-kharraz-mcheick.pdf>

Scope d'un objet

Les objets créés dans les JSP, les servlets ou les managed beans JSF, que nous verrons par la suite, ont une certaine portée (ou champ d'application), c'est-à-dire une certaine durée de vie. Les différents scopes sont les suivants :

- Page : durée de vie très courte. Les objets ne sont accessibles que dans la page où ils sont définis.
- Request : durée de vie courte. Les objets sont accessibles pendant la durée de la requête, c'est-à-dire entre le moment où elle est interceptée par le conteneur et le moment où la réponse est envoyée.
- Session : durée de vie longue. Les objets sont accessibles pendant toute la durée de la session, par exemple entre le moment où un utilisateur se connecte à un site puis se déconnecte.
- Application : durée de vie très longue. Les objets sont accessibles pendant toute la durée de vie de l'application.

Le langage d'expression

Avant de décrire JSTL, il faut introduire le langage d'expression utilisé dans les JSP. Le langage d'expression, ou *Expression Language* (EL), permet de manipuler des données plus simplement qu'avec les scriptlets (`<% %>`).

Une expression est de la forme suivante :

```
${ exp }
```

La chaîne `exp` correspond à l'expression à interpréter. Une expression peut être composée de plusieurs termes séparés par des points (notation pointée). Ainsi, pour accéder à la propriété `name` de notre entity `Product`, on utilise la syntaxe suivante :

```
${ product.name }
```

En fait, ce n'est pas l'attribut `name` qui est appelé, mais la méthode `getName()` puisque les accesseurs sont utilisés par introspection pour accéder aux propriétés d'un objet. Cette syntaxe remplace celle que nous avons vue précédemment :

```
<%= product.getName() %>
```

JSTL

En utilisant le design pattern MVC, on sépare les traitements de la présentation, c'est-à-dire le code Java, du code HTML. Cependant, il est souvent nécessaire d'effectuer de la logique d'affichage. Par exemple, imaginons les cas de figures suivants : « si le client est connecté, alors j'affiche un lien dans l'en-tête de ma page », ou, « tant que la fin de ma liste d'objets n'est pas atteinte, je la parcours ». Dans ce genre de cas, il faut avoir recours aux scriptlets pour rajouter des `if` et des `for`. Pour pallier ce problème, il est possible d'utiliser JSTL (*JSP Standard Tag Library*), ou bibliothèque de balises standards pour JSP.

Le but avoué de JSTL est l'utilisation de balises XML afin d'éviter de placer du code Java dans les JSP. C'est donc un ensemble de balises, regroupées dans quatre bibliothèques, qui proposent des fonctionnalités courantes comme :

- la bibliothèque `Core` : itération, condition, gestion des URL ;
- la bibliothèque `XML` : manipulation de données en provenance d'un document XML, transformation XSLT ;
- la bibliothèque `I18n` : internationalisation des messages, des dates, des nombres ;

- la bibliothèque Database : accès aux bases de données, définition de sources de données, exécution de requêtes SQL.

Dans le cadre du développement de l'application YAPS Pet Store, seule une partie de la bibliothèque Core présente un intérêt. Si vous souhaitez approfondir cette vaste spécification, reportez-vous aux références suivantes.

La bibliothèque Core contient les balises pour afficher des messages ou des attributs d'objets (out), pour conditionner une partie de la page (if, choose) ou pour itérer (forEach). Voyons dans l'exemple ci-après comment afficher une liste de produits en utilisant JSTL.

Exemple de JSP contenant des balises JSTL

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%> ❶

<%@ page import="com.yaps.petstore.entity.catalog.Product" %>
<%@ page import="java.util.List" %>
<TABLE>

  <jsp:useBean id="products" class="List<Product>"
               scope="request"/>

  <c:forEach var="product" items="${products}"> ❷
    <TR>
      <TD>
        <A href=""><c:out value="${product.name}"/></A> ❸
        <BR><c:out value="${product.description}"/>
      </TD>
    </TR>
  </c:forEach> ❷
</TABLE>
```

APPROFONDIR JSTL

JavaServer Pages Standard Tag Library :

- ▶ <http://java.sun.com/products/jsp/jstl/index.jsp>
- ▶ <http://java.sun.com/products/jsp/taglibraries.html>

📖 Shawn Bayern, *JSTL in Action*, Manning, 2002

📖 Sue Spielman, *JSTL – Practical Guide for JSP Programmers*, MK Publishers, 2003

❖ Si la JSP utilise un ou plusieurs tags de la bibliothèque Core, il faut le déclarer avec une directive `taglib`.

❖ Grâce au design pattern MVC, la servlet invoque un EJB et transmet la liste des produits à la JSP qui peut ensuite y accéder grâce à la balise `jsp:useBean`.

❖ La balise `forEach` itère la liste d'objets

❖ La balise `out` affiche la valeur retournée par l'expression `${product.name}`.

L'exemple précédent mélange les directives JSP (`jsp:useBean`) avec les balises JSTL (`forEach`, `out`) et le langage d'expression. Pour utiliser la bibliothèque Core de JSTL, il faut la déclarer dans la page ❶ à l'aide d'une directive JSP. Cette ligne de code nous informe de l'URL de la bibliothèque, mais aussi du préfixe utilisé dans la page. Toutes les balises préfixées par la lettre `c` correspondent à la bibliothèque Core. Ensuite, on peut parcourir ❷ la liste des produits avec la balise `forEach`. Celle-ci prend en paramètre la liste d'objets (`products`) puis définit un index (`product`) sur lequel itérer. On affiche alors le nom du produit et sa description ❸ à l'aide de la balise `out`. Comme vous pouvez le constater, le code se limite à des balises XML, le rendant beaucoup plus simple à lire et à comprendre.

JSTL Les balises personnalisées

JSTL possède une API qui permet de créer ses propres balises (*Custom Tags*). Écrites en Java et décrites en XML, elles sont ensuite regroupées dans une bibliothèque (*taglib*) pour être réutilisées dans des JSP. Cette extension de la technologie JSP est apparue à partir de la version 1.1 de la spécification.

OUTILS Les frameworks web

Struts est un des premiers frameworks Open Source pour développer des applications web basé sur le découpage MVC.

► <http://struts.apache.org/>

Spring MVC est le composant de navigation web du framework Spring. Son concept d'inversion de contrôle (IoC) permet notamment de construire une architecture web avec des couches indépendantes les unes des autres.

► <http://springframework.org/go-webflow2>

Tapestry est un autre ordonnanceur respectant le paradigme MVC.

► <http://tapestry.apache.org/>

APPROFONDIR JSF

► <http://java.sun.com/javaee/jaserverfaces/>

► <http://www.jsfcentral.com/>

📖 Kito Mann, *JavaServer Faces in Action*, Manning, 2005

RETOUR D'EXPÉRIENCE

La séparation des traitements et de la présentation dans les équipes

Nous venons de voir l'importance du design pattern MVC et de la répartition des responsabilités entre traitements et présentation. Ce découpage se fait aussi naturellement dans une équipe de développement. On y retrouve ainsi des développeurs Java s'occupant de traitements et des designers concevant la charte graphique de l'application, c'est-à-dire les pages web. Ces deux technologies, bien que complémentaires, ont du mal à coexister chez la même personne. Il est difficile de demander à un développeur Java de maîtriser l'art graphique et inversement. De plus, un designer risque de rencontrer des difficultés dans la manipulation de pages JSP lorsque celles-ci contiennent du code Java, puisqu'il ne travaille habituellement qu'avec des langages de balises. JSTL présente l'avantage de permettre au designer de continuer à utiliser des balises XML pour la construction de ses pages. Les taglibs (et Custom Tags) créent une nouvelle couche d'abstraction qui facilite la communication entre les deux langages.

JSF

Toutes les technologies que nous venons de voir sont nécessaires pour développer une application web et se complètent mutuellement. Par contre, rien dans les spécifications ne nous oblige à utiliser un modèle de programmation particulier (MVC par exemple). La navigation entre pages n'est pas non plus spécifiée, il est donc possible de la créer de différentes manières plus ou moins élégantes (en dur dans les servlets, via un fichier de configuration, etc.). Il manquait un modèle un peu plus rigide permettant de savoir clairement dans quel composant développer les traitements, où concentrer la présentation et comment naviguer entre les pages. Ce modèle est apparu avec JSF.

Java Server Faces permet de développer des applications web en bénéficiant de concepts déjà éprouvés par Java et Java EE (composants graphiques Swing, modèle de programmation événementiel, JSP, servlets, JSTL, langage d'expression), et par les apports d'autres framework Open Source tel que Struts.

JSF ne remplace pas les autres technologies web, il les utilise et les complète.

- Les servlets constituent le fondement de la technologie web et jouent le rôle de contrôleur du modèle MVC. La spécification précise aussi comment packager et déployer une application.
- Les JSP permettent de générer la partie graphique en utilisant des langages tels que HTML ou CSS (*Cascading Style Sheets*).

- Les balises JSTL simplifient le développement des pages JSP en ajoutant de la logique de présentation au format XML.
- Les langages d'expressions (EL et UEL que nous verrons par la suite) permettent d'accéder simplement aux objets.

À tout cela, JSF apporte plusieurs fonctionnalités destinées à résoudre les problèmes inhérents à la programmation web. JSF se compose d'un ensemble d'API fournissant notamment :

- une séparation nette entre la couche de présentation et les autres couches ;
- une librairie de composants graphiques ;
- un mode de programmation événementiel pour ces composants ;
- la navigation entre pages ;
- le traitement des formulaires et leur validation ;
- la gestion des exceptions et l'affichage de messages d'erreur ;
- la conversion des types primitifs provenant des données d'applications vers des objets de plus haut niveau (String vers Objets) ;
- la gestion de clients hétérogènes (HTML, WML, XML...) ;
- l'indépendance des protocoles (HTTP, WAP...) ;
- la création ou l'enrichissement de composants graphiques (custom) ;
- la gestion de l'état des composants entre requêtes ;
- la différence de comportement entre navigateurs.

Ce livre n'a pas la prétention de couvrir toutes les particularités de JSF. Nous ne nous attarderons que sur les balises graphiques, les traitements et la navigation entre pages.

Les balises JSF

JSF fournit aux développeurs une large palette d'outils leur permettant d'implémenter des applications web en respectant un modèle basé sur des composants graphiques (liste déroulante, zone de saisie, tableau, etc.). Chaque composant réagit ensuite à un ensemble d'événements (clic, changement de texte, etc.).

Le support de la partie graphique reste toujours une page JSP dans laquelle on rajoute les balises JSF. La page est donc constituée de composants graphiques qui s'imbriquent les uns dans les autres (une page contient un formulaire, qui est constitué de zones de saisie et de boutons) formant ainsi un arbre. Ces balises sont ensuite interprétées pour être transformées en HTML pour une application web, ou en WML pour une application destinée aux PDA.

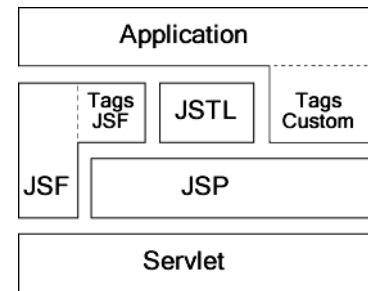


Figure 7-2
Technologies autour de JSF

ARCHITECTURE Le pattern observateur

Lorsqu'on parle de programmation événementielle, on pense souvent au pattern observable. Couramment utilisé dans les interfaces graphiques (c'est le cas de Swing et de JSF), ce pattern permet d'alerter des objets intéressés par un changement d'état (le contenu d'une zone de saisie qui change, un clic sur un bouton, etc.). Des objets demandent à observer (à recevoir une notification) le changement d'autres objets.

- ▶ <http://www.design-patterns.fr/Observateur.html>
- ▶ <http://c2.com/cgi/wiki?ObserverPattern>

WAP et WML

Le *Wireless Application Protocol* (WAP) est un protocole de communication dont le but est de permettre l'accès à Internet à l'aide d'un terminal mobile (par exemple un téléphone portable, un PDA, etc.). WML, *Wireless Markup Language*, est son langage de balises.

ARCHITECTURE Le design pattern composite

Lorsqu'on parle de structure en arbre, on évoque souvent le design pattern composite. Ce design pattern détermine une représentation en arbre d'une structure de données. En général, une composition est une collection d'objets, et tout objet peut être une composition (un nœud) ou un objet primitif (une feuille).

- ▶ <http://c2.com/cgi/wiki?CompositePattern>

Bibliothèque Core.	▶
Bibliothèque HTML.	▶

JSF utilise deux librairies de balises, HTML et Core, définies dans les pages JSP par les directives suivantes :

JSP déclarant les balises JSF

▶	<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
▶	<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

Cette déclaration implique que les balises Core seront préfixées par f (<f:view>) et les balises HTML par h (<h:commandLink>).

JSF Le rendu des balises

La bibliothèque de balises HTML est utilisée pour les rendus (*Renderer*) graphiques propres au langage HTML. Si vous voulez afficher une page pour un téléphone portable, une application Telnet, etc., il vous faudra utiliser une librairie différente. Il en existe plusieurs comme, par exemple, ADF d'Oracle.

▶ <http://www.oracle.com/technology/products/jdev/htdocs/partners/addins/exchange/jsf/>

Les balises HTML

Les balises HTML de JSF servent au rendu graphique. L'idée est de s'affranchir des balises HTML classiques et d'utiliser celles de JSF pour avoir des pages portables selon les navigateurs, répondant à des événements, et s'intégrant facilement aux traitements effectués par le contrôleur.

Dans les balises HTML de JSF, il y a tout d'abord celles qui vous permettent d'afficher ou de saisir du texte sous différentes formes. **outputText** ❶ affiche un message ou le contenu d'une variable, alors que **inputText** ❷ définit une zone de saisie et affecte la valeur à un objet. **inputSecret** ❸ est l'équivalent pour les zones de saisie de mot de passe.

Lorsque vous validez un formulaire qui comporte des données erronées, ou qu'une exception système intervient, il est nécessaire d'en informer l'utilisateur. La balise **message** ❹ répond à cette fonctionnalité en affichant les messages d'erreurs sur la page.

Balise JSF	Rendu graphique
<h:outputText value="#{cart.customer.lastname}"/> ❶	Smith
<h:inputText value="#{cart.customer.firstname}" /> ❷	
<h:inputSecret value="#{account.password}" ❸ maxLength="10" size="12"/>	
<h:message style="color: red"/> ❹ Login: <h:inputText value="#{account.login}"/> Pwd: <h:inputSecret value="#{account.pwd}"/> <h:commandButton value="Sign In" action="#{account.doSignIn}" type="submit"/>	Invalid login/password . Login: Pwd: Save

Une page comporte très souvent des boutons ❺, des liens ❻, des images ❼, ou des images ❽ qui se comportent comme des liens. Dans ce dernier exemple, on s'aperçoit qu'il est possible d'imbriquer des balises entre elles (**commandLink** contient une balise **graphicImage**).

Balise JSF	Rendu graphique
<code><h:commandButton value="Sign In" ⑤ action="#{account.doSignIn}" type="submit"/></code>	
<code><h:commandLink action="#{account.doSignIn}"> ⑥ <h:outputText value="Sign In"/> </h:commandLink></code>	<u>Sign In</u>
<code><h:graphicImage url="images/bird1.jpg"/> ⑦</code>	
<code><h:commandLink action="#{catalog.doShowItem}"> <h:graphicImage url="images/bird1.jpg"/> ⑧ </h:commandLink></code>	

Il existe bien d’autres composants graphiques disponibles dans JSF (comme les checkbox, les radiobouton, les listbox, etc.). Cet ouvrage ne se focalise que sur ceux qui seront utilisés dans l’application tels que la combobox ⑨ ou le tableau ⑩.

Arrêtons-nous un instant sur la balise `dataTable` ⑪. Elle permet d’itérer une liste (dans l’exemple, une liste de produits), de la formater sous forme de tableau (en utilisant la balise `column`) et de manipuler un index pour en obtenir des informations (ici la variable `product` désigne un produit unitaire). Pour afficher les attributs `name` et `description` du produit, il suffit d’utiliser la balise `outputText`.

Balise JSF	Rendu graphique								
<pre><h:selectOneMenu value="#{cart.creditCard.type}"> 9 <f:selectItem itemLabel="Visa"/> <f:selectItem itemLabel="Visa Gold"/> <f:selectItem itemLabel="Master Card"/> </h:selectOneMenu></pre>									
<pre><h:panelGrid columns="4"> 10 <h:outputText value="1" /> <h:outputText value="2" /> <h:outputText value="3" /> <h:outputText value="4" /> <h:outputText value="5" /> <h:outputText value="6" /> <h:outputText value="7" /> </h:panelGrid></pre>	<table><tr><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>5</td><td>6</td><td>7</td><td></td></tr></table>	1	2	3	4	5	6	7	
1	2	3	4						
5	6	7							

Balise JSF	Rendu graphique				
<pre><h:dataTable value="#{catalog.products}" var="product"> <h:column> ❶ <h:outputText value="#{product.name}"/> </h:column> <h:column> <h:outputText value="#{product.description}"/> </h:column> </h:dataTable></pre>	<table><tr><td>Bulldog</td><td>Excellent odorat</td></tr><tr><td>Caniche</td><td>Petit chien</td></tr></table>	Bulldog	Excellent odorat	Caniche	Petit chien
Bulldog	Excellent odorat				
Caniche	Petit chien				

JSF Les balises sont des classes

Les balises JSF sont définies par des classes Java qui se trouvent dans le paquetage `javax.faces.component`. Chaque classe hérite de `UIComponent` qui définit un certain nombre de méthodes pour interagir avec les managed beans, avoir un rendu graphique, convertir ou valider des données, etc.

JSF Convention de nommage des balises

Les balises peuvent être préfixées par ce que l'on veut. Il est pourtant commun d'utiliser la lettre `c` pour le Core JSP, `h` pour les balises HTML de JSF, et `f` pour le Core JSF.

JSF Créer ses propres convertisseurs

Un convertisseur est une classe Java qui doit implémenter les méthodes de l'interface `javax.faces.convert.Converter`. Suivez les étapes à l'adresse suivante :

► <http://www-128.ibm.com/developerworks/library/j-jsf3/>

Certaines balises JSF n'ont cependant pas de rendu graphique immédiat comme la balise `<h:form>` qui permet de gérer les formulaires.

Il faut bien comprendre que chacune de ces balises sera transformée en HTML pour ensuite constituer une page et être affichée par un navigateur. Par exemple, la balise JSF suivante :

```
<h:inputText value="#{cart.customer.firstname}" />
```

sera transformée en HTML comme ceci :

```
<input type="text" value="Smith" />
```

Les balises Core

Contrairement à la bibliothèque HTML, les balises Core n'ont pas de rendu graphique spécifique. Par convention, elles sont préfixées par la lettre `f`. Voici une liste non exhaustive de balises Core utilisées dans l'application.

Attachons-nous aux balises principales dans un premier temps. Les balises `view` ❶ et `subview` ❷ permettent à votre page de contenir d'autres balises JSF et de pouvoir être interprétées. Elles se retrouvent généralement en début et en fin de la JSP. La seule différence entre ces deux balises, se résume par le fait que `subview` est utilisée pour les fragments de pages, c'est-à-dire les en-têtes, menus ou bas de page.

La plupart des balises autorisent l'ajout de paramètres en incluant la balise `param` ❸. Pour ce qui est des listes de tout genre (combobox, listbox), on peut rajouter des libellés avec la balise `selectItem` ❹.

Lors de l'affichage de données, il est bien souvent nécessaire de les formater. Par exemple, on peut citer la conversion de dates au format `jj/mm/aaaa`, d'un entier avec deux chiffres après la virgule, etc. JSF possède deux balises standards pour vous faciliter la tâche `convertDateTime` ❺ et `convertNumber`. JSF vous permet aussi de créer vos propres convertisseurs.

Balise JSF	Description
<code><f:view></code> ❶ Page JSP utilisant des balises JSF <code></f:view></code>	Une page JSP voulant utiliser des balises JSF doit les placer dans cette balise. Elle se trouve généralement en début et en fin de page.
<code><f:subview></code> ❷ Sous-page JSP utilisant des balises JSF <code></f:subview></code>	Cette balise permet de créer des fragments de page qui pourront ensuite être englobés. Couramment utilisé pour les en-têtes, menus, ou bas de page.
<code><h:commandLink action="#{catalog.doFindProducts}"></code> <code><h:outputText value="Birds"/></code> <code><f:param name="categoryId" value="5"/></code> ❸ <code></h:commandLink></code>	Dans cet exemple, on rajoute le paramètre <code>categoryId</code> de valeur 5 à la balise <code>commandLink</code> .
<code><h:selectOneMenu value="#{cart.creditCard.type}"></code> <code><f:selectItem itemLabel="Visa"/></code> ❹ <code><f:selectItem itemLabel="Visa Gold"/></code> <code><f:selectItem itemLabel="Master Card"/></code> <code></h:selectOneMenu></code>	On ajoute des libellés dans une combobox.
<code><h:inputText value="#{account.customer.dateOfBirth}"></code> <code><f:convertDateTime pattern="dd.MM.yyyy"/></code> ❺ <code></h:inputText></code>	Convertit la date d'anniversaire au format <code>dd.MM.yyyy</code> .

Exemple de page JSP utilisant les balises JSF

Rien ne vaut un premier exemple de page web complète pour comprendre l'imbrication entre HTML, JSP et librairies JSF. Reprenons l'exemple de l'affichage de la liste des produits.

Extrait de JSP affichant les produits

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"%> ❶
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"%>

<f:view> ❷

  <h2>Products for category :
    <h:outputText value="#{catalog.category.name}"/> ❸
  </h2>

  <h:messages style="color: red"/> ❹
  <h:form>

    <h:dataTable value="#{catalog.products}" var="product"> ❺
      <h:column> ❻
        <h:commandLink action="#{catalog.doFindItems}"> ❼
          <h:outputText value="#{product.name}"/> ❽
          <f:param name="productId" value="#{product.id}"/>
        </h:commandLink>
```

APPROFONDIR Les balises JSF

Ce livre ne répertorie que certaines balises JSF. Vous pouvez en retrouver la totalité aux adresses suivantes :

- ▶ <http://horstmann.com/corejsf/jsf-tags.html>
- ▶ <http://www.exadel.com/tutorial/jsf/jsftags-guide.html>

- ❖ Importe les balises JSF à l'aide de la directive `taglib`.
- ❖ Les balises JSF doivent être à l'intérieur d'un `f:view`.
- ❖ On peut mêler balises HTML (`h2`) et balises JSF.
- ❖ Affiche les messages d'erreurs en cas d'exceptions.
- ❖ Itère la liste des produits.
- ❖ Déclare une colonne.
- ❖ Crée un lien hypertexte (`h:commandLink`) sur le nom du produit (`product.name`). Lorsqu'on clique sur le lien, un paramètre (`f:param`) est passé à l'action `doFindItems`.

Balise HTML de retour chariot.

Affiche la description du produit.

WEB Extensions des pages JSF

Les pages contenant des balises JSF sont des pages JSP. Elles possèdent donc l'extension `.jsp`.

HTML Un langage de balises

Le langage HTML est très riche et comporte beaucoup de balises. Dans l'application, nous n'en utiliserons qu'une infime partie, comme les balises de saut de ligne (`br`), de police de caractères (`h1`, `h2`, `strong...`) ou de structure (`body`, `head`, `html`). Vous pouvez retrouver la liste exhaustive à l'adresse suivante :

► <http://www.w3schools.com/tags/default.asp>

```
<br/>
<h:outputText value="#{product.description}"/> 9
</h:column>
</h:dataTable>
</h:form>
</f:view>
```

Pour utiliser les balises JSF, il faut obligatoirement importer les bibliothèques adéquates à l'aide des directives `taglib` ①, et les englober dans une balise `f:view` ②. Les balises Core seront alors préfixées par `f` et les balises graphiques par `h`.

Pour afficher le nom de la catégorie en titre de la page, on utilise la balise HTML `h2` conjointement avec la balise JSF `<h:outputText>` ③. Celle-ci est utilisée à différents endroits dans la page pour afficher le nom du produit ⑧ ainsi que sa description ⑨.

Le but étant d'afficher une liste de produits, on utilise la balise `<h:dataTable>` ⑤ pour itérer la collection (`catalog.products`) et la présenter sous forme de tableau ⑥. Pour représenter un lien HTML, on utilise la balise `<h:commandLink>` ⑦. Celle-ci possède un paramètre `action` qui permet, comme nous le verrons par la suite, d'appeler une méthode lorsqu'on clique sur le lien. En cas d'erreur, un message d'exception est affiché grâce à la balise `h:messages` ④.

Le résultat est le suivant : une page web qui affiche en titre le nom de la catégorie, puis itère la liste de ses produits en affichant leur nom, sous forme de lien hypertexte, et leur description (figure 7-3).



Figure 7-3

Page affichant les produits d'une catégorie

Voilà pour l'aspect graphique. Mais qu'en est-il des traitements ? Comment a-t-on réussi à obtenir une liste de produits ? De plus, dans presque tous les exemples que nous venons de voir, les balises utilisent fréquemment le symbole dièse `#`. Quelle est son utilité ?

Le langage d'expression unifié

Avant d'aborder la partie traitement, nous devons d'abord nous attarder sur le langage d'expression unifié et parler un peu du passé. Le langage d'expression (EL) que nous avons vu précédemment, est entré dans la spécification JSP 2.0. Il permet d'accéder facilement aux attributs d'un objet.

```
${ product.name }
```

Alors que la spécification JSP 2.0 n'était pas encore terminée, JSF 1.0 est apparu avec ses composants graphiques, ses convertisseurs, son mode de programmation événementielle et son langage d'expression.

```
#{ product.name }
```

La raison de la création de ce nouveau langage (# au lieu de \$), est principalement due au fait que le JSP EL ne satisfaisait pas le modèle JSF. En effet, JSP EL évalue une expression à la volée, c'est-à-dire que le conteneur résout l'expression au moment où il interprète la page, puis renvoie le résultat. JSF utilise un mode différé puisque l'expression peut être convertie, validée, répondre à un événement (un clic de souris, une valeur qui change) avant d'être évaluée. Les deux langages ont donc leur raison d'exister séparément.

Itération avec JSP EL

```
<c:forEach var="product" items="${products}"> ❶
  <h:inputText value="#{product.name}"/> ❷
</c:forEach>
```

Cet exemple itère une liste de produits `${products}` à l'aide d'une balise JSTL `forEach`, ❶ et utilise le langage d'expression JSP EL (`$`). L'expression `${products}` est donc évaluée immédiatement lors de l'interprétation de la page. Qu'en est-il de la variable `product` ? Lorsque la balise JSF `inputText` essaie d'évaluer l'expression `#{product.name}` ❷ en différé, il est trop tard, la variable n'est plus accessible. L'utilisation des deux langages en même temps peut donc faire apparaître des erreurs.

Pour résoudre ce problème, les spécifications JSP 2.1 et JSF 1.2 ont unifié leur langage. Les deux EL continuent à exister l'un à côté de l'autre, chacun avec sa spécificité (évaluation à la volée et en différé), mais peuvent désormais être utilisés conjointement, ce qui n'était pas le cas auparavant. Grâce au langage d'expression unifié (UEL), il est possible d'écrire le code suivant :

APPROFONDIR UEL

Unified Expression Language :

- ▶ <http://java.sun.com/products/jsp/reference/techart/unifiedEL.html>
- ▶ <http://java.sun.com/developer/technicalArticles/J2EE/jstl/>

JSF Cycle de vie d'une page

Le fait que l'évaluation soit différée est dû au cycle de vie de la page JSF qui est bien plus compliqué qu'une page JSP. Nous ne rentrerons pas dans le détail, mais il faut savoir que lorsque les balises de la page sont interprétées et exécutées, un arbre d'objets Java (les classes dérivées de `javax.faces.component.UIComponent`) est créé côté serveur. La racine de cet arbre est un objet `javax.faces.component.UIViewRoot`. Cet arbre est ensuite transformé (phase de rendu) dans un flux propre à la technologie du client : HTML, WML, XML, etc. Lorsque le client soumet la vue courante, le flux correspondant est décodé pour reconstituer l'arbre de composants associés.

Itération avec JSF UEL

```
<c:forEach var="product" items="#{products}"> ❸
  <h:inputText value="#{product.name}"/>
</c:forEach>
```

La balise JSTL `forEach` peut utiliser UEL pour évaluer la liste des produits ❸.

JSF La gestion événementielle

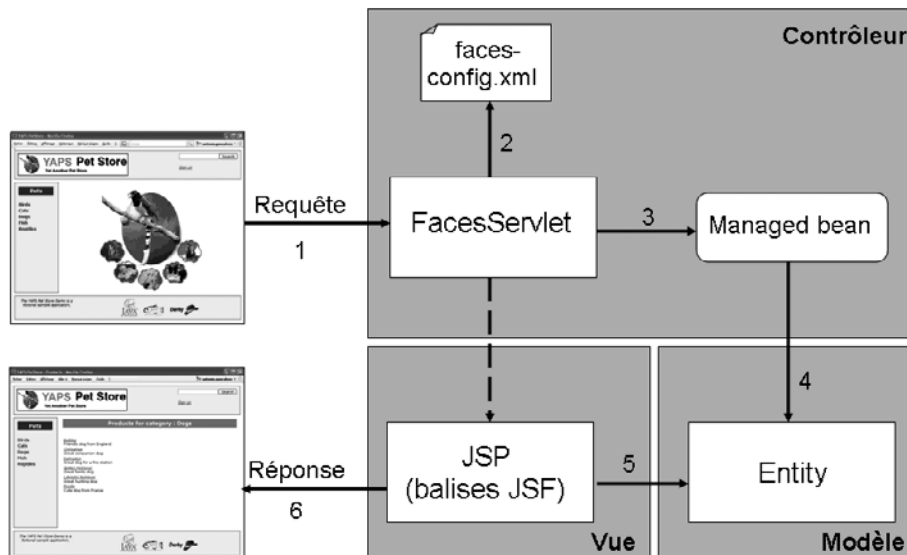
Le modèle MVC que nous décrivons ici est légèrement simplifié. En effet, JSF utilise un mode de programmation événementiel (similaire à Swing). Il y a donc toute une panoplie d'actions handler répondant à des événements: `ActionListener` lorsqu'on clique sur un bouton ou un lien, ou `ValueChangeListener` lorsque le contenu d'une zone de saisie change. Nous n'aborderons pas ces concepts avancés dans ce livre.

► http://www.oracle.com/technology/oramag/oracle/04-mar/o24dev_jsf.html

Traitements et navigation

L'architecture de JSF est basée sur le design pattern MVC. Comme nous l'avons vu précédemment dans MVC, les servlets sont utilisées pour intercepter une requête, effectuer un traitement, puis déléguer l'affichage à une JSP. Les servlets s'occupent donc de récupérer les informations de la requête, d'effectuer des traitements (appeler des EJB stateless par exemple), mais aussi de naviguer entre pages. Dans ce modèle, on a donc une relation 1:1 entre une servlet et une JSP, ce qui, pour créer un site complet, multiplie considérablement le nombre de servlets. De plus, les règles de navigation sont codées en dur, ce qui rend les modifications compliquées.

Figure 7-4
Architecture MVC de JSF



Pour rendre le développement d'une application web plus modulaire et plus souple, JSF décore tous ces éléments (traitement, présentation, navigation) et enrichit le pattern MVC. Le modèle est toujours représenté par les entités ❺. Par contre, il n'y a qu'un contrôleur unique (le `FacesServlet`) qui intercepte les requêtes HTTP ❶, une navigation configurable par fichier

XML ❷, des traitements délégués à des managed beans ❸ qui manipulent le modèle ❹ et un rendu graphique utilisant les balises JSF ❺.

La FacesServlet

Le contrôleur JSF est le point d'entrée de toutes les requêtes HTTP. Il est constitué d'une servlet unique nommée `FacesServlet`. La `FacesServlet` reçoit les requêtes de la part des clients web, et exécute un certain nombre d'étapes pour préparer la réponse qui sera affichée par le navigateur. Les actions et la navigation entre pages sont paramétrables par fichier XML (`faces-config.xml`).

Pour que cette servlet puisse intercepter toutes les requêtes concernant JSF, elle doit être déclarée dans le fichier `web.xml` de l'application.

Déclarer les servlets dans le fichier `web.xml`

Le fichier `web.xml` est un fichier extrêmement important dans une application Java EE. Aussi appelé descripteur de déploiement, il contient les caractéristiques et paramètres de l'application, ce qui inclut la description des servlets utilisées et leurs différents paramètres d'initialisation. Chaque servlet doit être déclarée dans l'élément XML `<servlet>` de la manière suivante :

- `<servlet-name>` est le nom interne de la servlet, nom qui l'identifie de façon unique.
- `<servlet-class>` est la classe Java associée à la servlet.
- `<load-on-startup>` demande que la servlet soit chargée dès le démarrage du serveur.
- `<servlet-mapping>` effectue le lien entre l'URL et la servlet.
- `<url-pattern>` est l'URL permettant de faire le lien avec la servlet. Dans notre cas, chaque URL ayant comme suffixe `.faces`, invoquera automatiquement la `FacesServlet`.

Extrait du fichier `web.xml`

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>*.faces</url-pattern>
</servlet-mapping>
```

JSF Les implémentations

JSF n'est qu'une spécification. Il vous faut donc utiliser une implémentation pour pouvoir exécuter des pages et traitements JSF.

Il existe plusieurs implémentations, plus ou moins abouties :

- MyFaces d'Apache
 - <http://myfaces.apache.org/>
- Mojarra de Sun (GlassFish)
 - <https://javaserverfaces.dev.java.net/>

Consultez la matrice suivante, comparant les implémentations :

- <http://www.jsfmatrix.net/>

◀ On déclare la `FacesServlet`.

◀ La servlet est invoquée lorsqu'une ressource possède l'extension `faces`.

APPROFONDIR **Un bean**

Un bean est une classe Java qui respecte la spécification JavaBeans : un constructeur vide, des méthodes publiques de lecture et de modification de ses attributs (getXXX et setXXX), etc.
► <http://java.sun.com/beans/docs/>

Managed bean.	►
La référence de l'EJB est injectée.	►
Attributs privés du managed bean accessibles uniquement via les méthodes publiques get et set.	►
Méthode du bean.	►
Alias utilisé pour la navigation.	►

JSF Nommer les managed beans

Java EE n'impose pas de règles de nommage pour ses composants. Toutefois, il est commun de suffixer les stateless beans par Bean. En ce qui concerne les managed beans, pour ne pas répéter le suffixe Bean, on peut utiliser Controller ou Action.

Alias qui sera utilisé dans les pages.	►
Classe Java devant être prise en compte par JSF.	►
Champ d'application du bean.	►

Le managed bean

La FacesServlet délègue les traitements et les appels aux managed beans. Un managed bean est une classe dont le cycle de vie est géré par JSF. En fait, JSF instancie le bean en fonction de son champ d'application (scope), le stocke dans le contexte JSF, et l'invoque lorsque nécessaire. Ci-après un exemple de managed bean :

Exemple de managed bean

```
public class CatalogController
{
    @EJB ❶
    private CatalogLocal catalogBean; ❷

    private String keyword; ❸
    private List<Item> items;

    public String doSearch() { ❹
        items = catalogBean.searchItems(keyword); ❺
        return "items.found" ❻
    }
    // Getters & setters des attributs
}
```

Ce managed bean CatalogController déclare deux attributs privés ❸ et une méthode publique ❹. Celle-ci appelle la méthode searchItems ❺ du stateless session bean (CatalogBean) qui effectue réellement les traitements métier. La chaîne de caractères ❻ retournée par la méthode est utilisée pour la navigation entre pages. Nous l'expliquerons par la suite.

Comme vous pouvez le voir, cette classe est tout à fait simpliste. En fait, pour devenir managed bean et être pris en compte par JSF, il faut tout simplement la déclarer dans le fichier faces-config.xml.

Exemple de déclaration de managed bean

```
<managed-bean>
  <managed-bean-name>catalog</managed-bean-name>
  <managed-bean-class>
    com.yaps.petstore.jsf.CatalogController
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Revenons sur l'extrait de code du managed bean CatalogController. Pour appeler l'EJB, le managed bean peut utiliser son interface locale puisqu'ils sont tous deux déployés dans le même fichier .ear. Par contre,

le code n'utilise pas l'API JNDI pour retrouver l'interface. Une des nouveautés de Java EE 5 est le système d'injection. Le code précédent déclare une interface locale ❷, et l'annotation @EJB en injecte la référence ❶.

L'injection

La plupart des applications ont besoin de ressources externes (source de données, interface d'EJB, file d'attente JMS ou service web). En J2EE 1.4 le client doit explicitement déclarer cette dépendance dans des descripteurs de déploiement XML et en obtenir une référence à l'aide de JNDI.

En Java EE 5, ces références sont instanciées par le conteneur lui-même, puis injectées dans les objets managés. Ce mécanisme est appelé injection (en anglais *Dependency Injection*, ou *Inversion of Control* – IoC). Cette injection de dépendance ne peut être utilisée que pour des objets pris en compte par le conteneur.

L'application Swing étant distante et exécutée en dehors d'un conteneur client (ACC), elle ne peut pas utiliser ce mécanisme d'injection et doit avoir recours aux lookups JNDI. Les managed beans, quant à eux, s'exécutent à l'intérieur du conteneur web et peuvent donc utiliser le mécanisme d'injection. JNDI est, bien sûr, toujours sollicité mais de façon plus discrète. Le développeur n'a plus besoin de manipuler directement l'API JNDI.

L'injection de l'EJB se fait grâce à l'annotation @javax.ejb.EJB.

Code de l'annotation @javax.ejb.EJB

```
package javax.ejb;

@Target(value={TYPE, METHOD, FIELD}) @Retention(value=
RUNTIME)
public @interface EJB {
    String name() default "";
    String description() default "";
    String beanName() default "";

    Class beanInterface()
    String mappedName() default "";
}
```

En annotant un attribut avec @EJB, le conteneur l'initialise automatiquement avec la référence vers l'EJB (ou plus précisément son interface locale). Cette initialisation se produit avant qu'aucune autre méthode ne soit appelée. Notre code ne contient alors plus aucune API technique

ARCHITECTURE Principe de l'injection

Martin Fowler est à l'origine du terme *Dependency Injection*, ou injection. Au lieu de faire un lookup de l'interface de l'EJB, celle-ci est injectée par le conteneur dans le managed bean.

► <http://www.martinfowler.com/articles/injection.html>

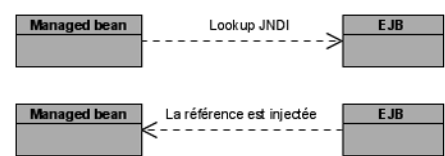


Figure 7-5 L'injection

- ❖ S'applique à une classe, une méthode ou un attribut.
- ❖ Nom JNDI de l'EJB utilisé.
- ❖ Description.
- ❖ Nom de l'EJB. Correspond au nom défini dans l'annotation @Stateless de l'EJB.
- ❖ Classe de l'interface retournée par le lookup.
- ❖ Référence à l'EJB spécifique au serveur d'applications. Non portable.

(JNDI), et la résolution des dépendances est fortement typée (les opérations de cast ne sont pas nécessaires).

La glue entre le managed bean et la page

D'un côté, nous avons des managed beans qui contiennent des attributs, des méthodes, et de l'autre nous avons des pages JSP qui affichent ces attributs et invoquent ces méthodes. Le lien entre les deux se fait par le langage d'expression. Prenons l'exemple d'un managed bean qui déclare un attribut keyword de type String :

Exemple de managed bean déclarant un attribut

Managed bean.	▶	<code>public class CatalogController</code>
Attribut du managed bean.	▶	<code>private String keyword; (...)</code>
Getter de l'attribut.	▶	<code>public String getKeyword() { return keyword; } }</code>

Pour afficher l'attribut keyword de ce managed bean, les pages utilisent l'expression `#{catalog.keyword}`. Le préfixe `catalog` fait référence à l'alias défini dans le fichier `faces-config.xml`. Ainsi, `#{catalog.keyword}` équivaut à invoquer `CatalogController.getKeyword()`.

Déclaration de l'alias dans le faces-config.xml

```
<managed-bean>  
  <managed-bean-name>catalog</managed-bean-name>  
  <managed-bean-class>  
    com.yaps.petstore.jsf.CatalogController  
  </managed-bean-class>  
</managed-bean>
```

La navigation entre pages

Autre élément pris en compte par JSF, la navigation entre les pages de l'application web. À l'aide de règles de navigation (*navigation rule*) décrites dans le fichier `faces-config.xml`, JSF va déterminer en fonction de la page courante quelle est la page suivante. Une règle de navigation s'écrit de cette manière :

Exemple de règle de navigation définie dans le faces-config.xml

À partir de cette page...	▶	<code><navigation-rule> <from-view-id>showaccount.jsp</from-view-id> ① <navigation-case></code>
---------------------------	---	--

```

<from-outcome>updateaccount</from-outcome> ❷
<to-view-id>/updateaccount.jsp</to-view-id> ❸
</navigation-case>
</navigation-rule>

```

- ◀ ... et pour cette clé de navigation...
- ◀ ... on se rend sur cette page.

Cette règle se lit de la façon suivante : « si je suis sur la page `showaccount.jsp` ❶ et que la clé de navigation est `updateaccount` ❷, alors je me rends à la page `updateaccount.jsp` ❸ ». La clé de navigation est la chaîne de caractères retournée par une méthode du managed bean (les méthodes retournent toutes une `String`).

Clé de navigation retournée par une méthode

```

public class AccountController {
    public String doUpdateAccount() {
        (...)
        return "updateaccount"; ❷
    }
}

```

Il est aussi possible d'utiliser des expressions régulières dans la définition des règles de navigation pour inclure un plus grand nombre de cas en une seule règle. L'exemple ci-après définit une règle qui permet à partir de n'importe quelle page ❹ d'aller vers `main.jsp` ❻ si la clé de navigation est `main` ❺.

Exemple de règle de navigation avec expression régulière

```

<navigation-rule>
  <from-view-id>*</from-view-id> ❹
  <navigation-case>
    <from-outcome>main</from-outcome> ❺
    <to-view-id>/main.jsp</to-view-id> ❻
  </navigation-case>
</navigation-rule>

```

- ◀ À partir de n'importe quelle page...
- ◀ ... et pour cette clé de navigation...
- ◀ ... on se rend sur cette page.

Deux composants JSF déclenchent des actions de navigation : un clic sur un bouton (balise `commandButton`) ou un lien hypertexte (`commandLink`). Lorsque l'utilisateur clique sur l'un ou l'autre, une clé de navigation (outcome) est renvoyée. Cette clé peut être statique, en dur dans le code de la page JSP, ou dynamique, calculée par une EL.

Navigation statique

On appelle navigation statique un enchaînement entre deux pages qui sera toujours le même. Un clic sur un bouton ou sur un lien hypertexte

déclenchera toujours la même destination. Pour illustrer ce propos, l'exemple suivant affiche un lien hypertexte avec une action ayant comme valeur `updateaccount`.

Exemple de navigation statique

```
<h:commandLink action="updateaccount">
  <h:outputText value="Edit Your Account Information"/>
</h:commandLink>
```

Lorsque l'utilisateur clique sur le lien HTML, la clé de navigation `updateaccount` est renvoyée au gestionnaire de navigation de JSF qui va rechercher une règle applicable (définie dans le fichier `faces-config.xml`) et déterminer la page suivante à afficher.

Navigation dynamique

Dans le cas de la navigation dynamique, la clé de navigation est inconnue au moment où la page est écrite. C'est donc le managed bean qui va retourner cette clé via l'appel de l'une de ses méthodes. Dans l'exemple suivant, la valeur de l'action n'est pas connue. On utilise le langage d'expression pour appeler la méthode `doUpdateAccount` du managed bean `account` :

Exemple de navigation dynamique

```
<h:commandLink action="#{account.doUpdateAccount}">
  <h:outputText value="Edit Your Account Information"/>
</h:commandLink>
```

La valeur de retour de l'action doit être de type `String`, et doit être référencée par une règle de navigation.

Le managed bean calcule la clé de navigation

```
public class AccountController {
    public String doUpdateAccount() {
        if (...)
            return "success";
        else
            return "failure";
    }
}
```

JSF Clé de navigation à null

Les méthodes des managed beans doivent retourner une clé de navigation de type `String`. Cette clé doit correspondre à une règle de navigation définie dans le fichier `faces-config.xml`. Cette valeur peut aussi être égale à `null`. Ce cas provoque le réaffichage de la page courante.

La valeur de la clé de navigation varie en fonction du résultat de la méthode `doUpdateAccount` (ici `"success"` ou `"failure"`). La navigation qui en résulte est donc variable (dynamique) et l'utilisateur sera redirigé soit vers une page, soit vers une autre. Bien sûr, il faudra déclarer ces nouvelles règles dans le fichier `faces-config.xml`.

Comment développer une application web avec JSF

Nous venons de définir de nombreux concepts. Il est temps de les assembler pour comprendre comment créer une application web. Prenons l'exemple de la recherche des articles. Dans le cas d'utilisation « Rechercher un article » du premier chapitre, l'application web doit pouvoir afficher la liste des articles en se basant sur une chaîne de caractères saisie par un internaute. Voici le fonctionnement général (figure 7–6).

La chaîne de caractères est saisie dans une balise `inputText`. Lorsque l'internaute clique sur le bouton `Search` (`commandButton`), la méthode `doSearch` du managed bean `CatalogController` est invoquée ❶. Celui-ci délègue le traitement à l'EJB stateless `CatalogBean` qui lui retourne une liste d'entités `Item` ❷. Grâce au fichier `faces-config.xml`, la navigation se fait vers la page `searchresult.jsp` ❸ qui affiche la liste dans une balise `dataTable` ❹.

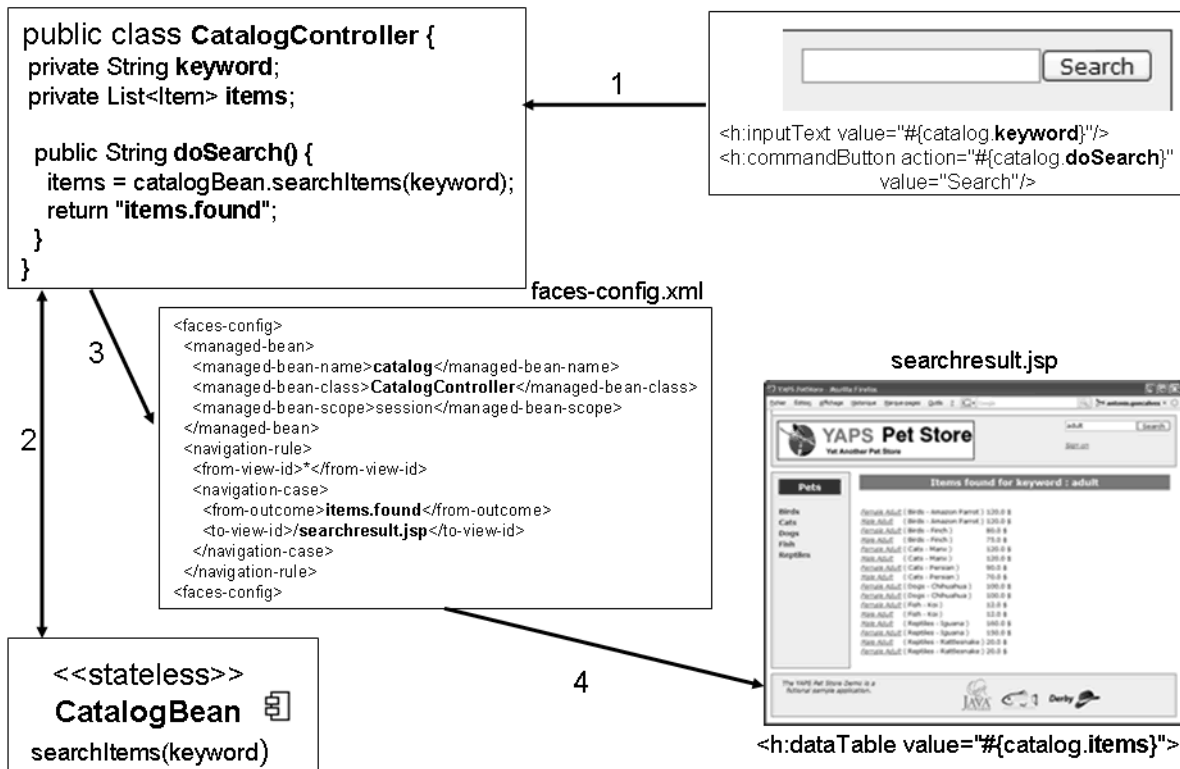


Figure 7–6 Recherche d'articles

Affiche une zone de saisie.	➤
Le bouton intitulé "Search" déclenche une action.	➤

Web HTTP, un protocole sans mémoire

Rappelons que le protocole HTTP est non connecté, c'est-à-dire qu'il ne garde aucune information concernant l'appelant entre deux requêtes. La session HTTP (classe HttpSession) est le seul moyen qui permet de s'affranchir de ce problème en maintenant, au fil des pages accédées par l'internaute, une transmission des données, le tout pour une période limitée : la durée de vie de la session.

Alias utilisé pour invoquer le managed bean.	➤
La classe du managed bean.	➤

Managed bean.	➤
La référence de l'EJB est injectée.	➤
Les attributs sont accessibles dans les pages via le langage d'expression. Par exemple : <code>#{catalog.keyword}</code> .	➤
Méthode recherchant les articles.	➤
Passe la chaîne de caractères à l'EJB.	➤
Alias utilisé pour la navigation.	➤

Voyons maintenant en détail comment ces enchaînements se font. Tout d'abord, concentrons-nous sur la saisie de la chaîne de caractères et l'appel au managed bean. Le code ci-après affiche une zone de saisie ❶ et un bouton ❷. Lorsqu'on clique sur le bouton, l'action `catalog.doSearch` est invoquée (attribut action de la balise `commandButton`).

Extrait de la JSP permettant la saisie de la chaîne de caractères

```
<h:inputText value="#{catalog.keyword}"/> ❶
<h:commandButton action="#{catalog.doSearch}" ❷
value="Search"/>
```

Le suffixe `catalog` ❶ ❷ que vous voyez dans cet extrait de code, fait référence au managed bean `CatalogController` ❹. Cet alias ❸ se fait par le fichier de configuration `faces-config.xml`. JSF crée une instance de la classe `com.yaps.petstore.jsf.CatalogController` ❹ qu'il nomme `catalog` ❸. La portée de cet objet est `session` ❺, c'est-à-dire que le managed bean et ses attributs pourront être utilisés tout au long de la session de l'utilisateur.

Extrait du `faces-config.xml` déclarant le managed bean

```
<managed-bean>
  <managed-bean-name>catalog</managed-bean-name> ❸
  <managed-bean-class>
    com.yaps.petstore.jsf.CatalogController ❹
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope> ❺
</managed-bean>
```

Lorsque l'évaluation de l'expression `#{catalog.doSearch}` ❷ se fait, la méthode `doSearch` ❹ du managed bean est invoquée. La zone de saisie ❶, quant à elle, est mappée avec l'attribut `keyword` ❷. Ainsi, lorsqu'on saisit une chaîne de caractères et que l'on clique sur le bouton, la valeur est stockée dans la variable `keyword`, et la méthode `doSearch` est invoquée.

Extrait du managed bean `CatalogController`

```
public class CatalogController
{
  @EJB
  private CatalogLocal catalogBean; ❹
  private String keyword; ❷
  private List<Item> items; ❸

  public String doSearch() { ❹
    items = catalogBean.searchItems(keyword); ❺
    return "items.found" ❻
  }
}
```

La méthode `doSearch` ⑨ appelle l'EJB stateless `CatalogBean` au travers de son interface locale `CatalogLocal` ⑥ en lui passant la chaîne de caractères `keyword` en paramètre ⑩. Le résultat de la méthode `searchItems`, qui est une liste d'entités `Item`, est alors stocké dans l'attribut `items` ⑧ du managed bean.

Les traitements sont maintenant terminés et la méthode retourne la clé de navigation `"items.found"` ⑪. Le fichier `faces-config.xml` fait coïncider cette clé de navigation ⑪ ⑫ avec la page `searchresult.jsp` ⑬ qui affichera la liste des articles.

Extrait du `faces-config.xml` déclarant la navigation

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>items.found</from-outcome> ⑫
    <to-view-id>/searchresult.jsp</to-view-id> ⑬
  </navigation-case>
</navigation-rule>
```

Le contrôleur de JSF se charge d'appeler la page `searchresult.jsp`. Celle-ci n'a plus qu'à itérer ⑭ la liste des articles contenus dans l'attribut `items` ⑧ du managed bean et en afficher les informations ⑮.

Extrait de la page `searchresult.jsp` affichant la liste des articles

```
<h:dataTable value="#{catalog.items}" var="item"> ⑭
  <h:column>
    <h:outputText value="#{item.name}"/> ⑮
  </h:column>
  <h:column>
    <h:outputText value="#{item.product.category.name}"/>
    <h:outputText value="#{item.product.name}"/> ⑮
  </h:column>
  <h:column>
    <h:outputText value="#{item.unitCost}"/> ⑮
  </h:column>
</h:dataTable>
```

◀ À partir de n'importe quelle page, on applique cette règle de navigation.

◀ Clé de navigation.

◀ Page de destination.

JSF Le contrôleur `FacesServlet`

Comme vous pouvez le voir dans cet exemple, nous ne manipulons pas directement la `FacesServlet`. La servlet est déployée dans le conteneur et intercepte les requêtes HTTP, sans que nous ayons à interagir avec.

◀ Itère la liste d'articles sous forme de tableau.

◀ Affiche le nom de l'article dans une colonne.

◀ Le langage d'évaluation permet d'obtenir les attributs des objets liés :
`item.getProduct().getCategory().getName()`.

◀ Affiche le prix de l'article.

RETOUR D'EXPÉRIENCE Développer une application web

Développer une application web n'est pas chose facile. Comme nous l'avons vu en introduction, les spécifications ont dû se succéder pour aboutir à JSF : les servlets ont laissé place aux JSP qui ont dû s'enrichir de bibliothèques de balises JSTL avant la venue de JSF. Même si JSF fédère ces spécifications, il est tout de même nécessaire de connaître plusieurs API et langages (HTML, Java). De plus, la glue (le lien) entre toutes ces technologies se fait par fichier XML et par l'utilisation de langages d'expressions (EL, UEL). La navigation entre pages est aussi un élément sensible d'une application web et peut s'avérer complexe pour une application de grande envergure. Sans parler de tous les problèmes auxquels doit faire face le développeur (protocole HTTP sans état, bouton retour des navigateurs...).

L'objectif de ce livre est de se concentrer uniquement sur Java EE 5, ce qui est déjà un vaste programme. J'ai donc sciemment laissé de côté tout framework Open Source qui serait venu rajouter un niveau de complexité supplémentaire à la lecture de ce livre. Ceci dit, je me dois de vous parler de JBoss Seam.

JBoss Seam (futur Web Beans, JSR n° 299) est un framework de développement d'applications web qui unifie et intègre JSF et les EJB3, entre autres. Il a été pensé pour éliminer la complexité inhérente aux applications web, et pour enrichir certaines lacunes liées au protocole HTTP. JBoss Seam offre de nombreuses facilités pour le développement d'applications et se présente de plus en plus comme une alternative aux développements web classiques.

► <http://www.jboss.com/products/seam>

► <http://www.jcp.org/en/jsr/detail?id=299>

L'application web YAPS Pet Store

L'application web YAPS Pet Store permet aux internautes de visualiser le catalogue et de rechercher des articles. L'internaute peut aussi se connecter en se créant un compte et devenir ainsi client de la société. Il peut alors mettre à jour les données de son compte.

Décorateurs

Avant de nous concentrer sur les fonctionnalités de l'application web, regardons son aspect graphique. Toutes les pages sont construites de la même manière et utilisent les mêmes éléments, c'est-à-dire (figure 7-7) :

- un en-tête (`header.jspf`) qui affiche le logo et le nom de la société ainsi que des liens pour se connecter et se déconnecter du site ;
- sur la gauche, une barre de navigation (`navigation.jspf`) liste toutes les catégories des produits ;
- un bas de page (`footer.jspf`) où se trouvent les logos des technologies utilisées (Java, GlassFish et Derby).

Au lieu de dupliquer ces éléments sur toutes les pages, on les inclut à l'aide de la directive `include` des JSP. Ci-après le patron d'une page qui nous montre comment inclure ces différents éléments ❶. La partie spécifique est contenue dans le corps de chaque page ❷.

GRAPHISME Cascading Style Sheet

L'application utilise le fichier `petstore.css` pour définir tous les aspects graphiques (couleurs, polices de caractères, etc.).

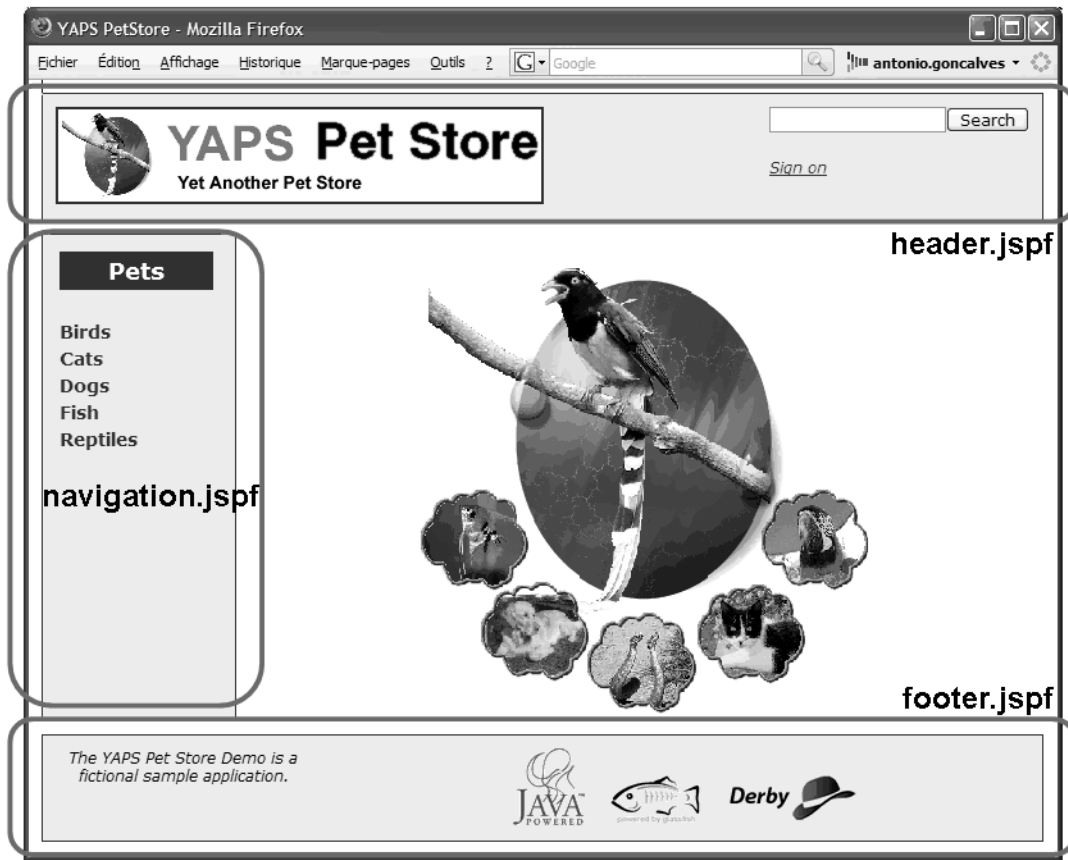


Figure 7–7
En-tête, menu
et pied de page

Extrait d'une page JSP

```
<f:view> ③
  <div id="header">
    <%@ include file="common/header.jspf" %> ①
  </div>
  <div id="sidebar">
    <%@ include file="common/navigation.jspf" %> ①
  </div>
  <div id="body">
    Corps de la page ②
  </div>
  <div id="footer">
    <%@ include file="common/footer.jspf" %> ①
  </div>
</f:view> ③
```

La page principale est englobée dans une balise `<f:view>` ③ alors que les fragments (`header.jspf`, `menu.jspf` et `footer.jspf`) doivent être englobés dans une balise `<f:subview>` ④.

Extrait de la page footer.jspf

```
<f:subview> 4
    <h:outputText value="The YAPS Pet Store Demo is a fictional
        sample application."/>

    
    
    
</f:subview> 4
```

La visualisation du catalogue

La visualisation du catalogue et la recherche d'articles sont détaillées dans les cas d'utilisation « Visualiser les articles du catalogue » et « Rechercher un article ». Un internaute peut visualiser le contenu du catalogue des animaux domestiques. Il peut aussi utiliser le mécanisme de recherche qui lui permet de saisir une chaîne de caractères et lui retourne une liste d'articles appropriés.

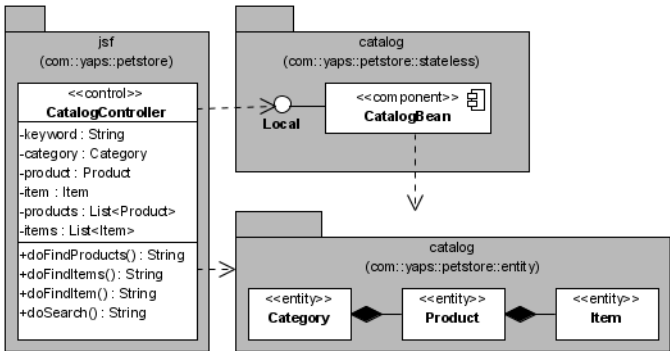
Le managed bean CatalogController

Le managed bean CatalogController permet de faire toutes les actions concernant le catalogue. Il fait le lien entre les pages JSP et l'EJB stateless CatalogBean qui manipule les données de l'application via les entités (Category, Product et Item). Le diagramme de classes ci-après montre ces relations (figure 7-8).

REMARQUE Simplification du code

Pour que les lignes de code qui suivent soient plus faciles à lire, certaines simplifications ont été faites. La gestion des exceptions, l'affichage des messages d'erreurs ou tout simplement les traces (API de logging) ne seront pas développés.

Figure 7-8
Diagramme du CatalogController
et de ses dépendances



Extrait du CatalogController

```
public class CatalogController
    @EJB
    private CatalogLocal catalogBean;
```

La référence de l'EJB est injectée dans l'attribut catalogBean.

```

private String keyword;
private Category category;
private Product product;
private Item item;
private List<Product> products;
private List<Item> items;
public String doFindProducts() {
    category=catalogBean.findCategory(
        getParamId("categoryId"));
    products = category.getProducts();
    return "products.displayed";
}

public String doFindItems() {
    product = catalogBean.findProduct(getParamId("productId"));
    items = product.getItems();
    return "items.displayed";
}

public String doFindItem() {
    item = catalogBean.findItem(getParamId("itemId"));
    return "item.displayed";
}

public String doSearch() {
    items = catalogBean.searchItems(keyword);
    return "items.found";
}
// Getters & setters des attributs
}

```

◀ Ces attributs sont manipulés par les pages JSP via le langage d'expression (exemple : `#{catalog.keyword}`)

◀ Cette méthode retourne les produits d'une catégorie.

◀ Cette méthode retourne les articles d'un produit.

◀ Pour un identifiant donné, cette méthode retourne un article.

◀ À partir d'une chaîne de caractères (keyword), cette méthode retourne une liste d'articles.

Vous remarquerez à plusieurs reprises l'appel à la méthode `getParamId`. Celle-ci permet de récupérer un paramètre passé par la page JSP. Par exemple, lorsqu'on clique sur un lien hypertexte et que l'on veut passer un paramètre au managed bean, on utilise la balise `<f:param>`. Dans l'exemple suivant, on affecte la valeur 5 au paramètre `catagoryId`.

Lien hypertexte utilisant la balise `<f:param>`

```

<h:commandLink action="#{catalog.doFindProducts}">
    <h:outputText value="Birds"/>
    <f:param name="categoryId" value="5"/>
</h:commandLink>

```

Lorsqu'on clique sur ce lien, la méthode `doFindProducts` du managed bean est invoquée. On peut ensuite récupérer ce paramètre grâce à la méthode `getParamId`.

Le managed bean récupère la valeur de ce paramètre

```

public String doFindProducts() {
    category=catalogBean.findCategory(getParamId("categoryId"));
    products = category.getProducts();
    return "products.displayed";
}

```


JSF Le contexte

Le contrôleur `FacesServlet` crée un objet `FacesContext` qui contient les informations nécessaires à l'exécution d'une requête utilisateur, c'est-à-dire les objets `ServletContext`, `ServletRequest` et `ServletResponse` qui sont fournis par le conteneur web.

REMARQUE La recherche d'articles

La recherche d'articles a été développée dans le paragraphe « Comment développer une application web avec JSF ». Nous ne l'aborderons donc pas ici.

À partir de n'importe quelle page (puisque chaque page contient le menu de navigation), si on rencontre la clé de navigation `products.displayed`, la page des produits est affichée.

La méthode `getParamId` utilise le contexte JSF (`FacesContext`) pour récupérer les paramètres passés à la requête HTTP (`getRequestParameterMap`). Ces paramètres sont stockés dans une map sous la forme clé/valeur. Il suffit ensuite de récupérer le paramètre qui nous intéresse (`map.get(param)`).

La méthode `getParamId`

```
protected Long getParamId(String param) {
    FacesContext context = FacesContext.getCurrentInstance();
    Map<String, String> map =
        context.getExternalContext().getRequestParameterMap();
    String result = map.get(param);
    return Long.valueOf(result );
}
```

Les pages web

La visualisation du catalogue utilise trois pages :

- affichage des produits d'une catégorie (`showproducts.jsp`) ;
- affichage des articles d'un produit (`showitems.jsp`) ;
- détail d'un produit (`showitems.jsp`).

La liste des catégories (« chat », « chien »...) est contenue dans le menu de gauche de chaque page, les rendant ainsi accessibles à travers tout le site.

La navigation

La navigation entre les pages est assez intuitive puisqu'elle suit la démarche des clients (figure 7-9). Pour acheter un animal, on commence par choisir la catégorie ❶ (« je voudrais acheter un chien ») puis le produit ❷ (« un caniche ») et enfin l'article lui-même ❸ (« un mâle adulte »). En face de chaque article, on affiche une image représentant l'animal, son nom, sa description ainsi que son prix. Comme le client peut vouloir acheter plusieurs articles, le menu de gauche lui permet à tout moment de naviguer dans les catégories.

La navigation entre ces pages est définie dans le fichier `faces-config.xml`.

Extrait du `faces-config.xml` concernant la navigation du catalogue

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>products.displayed</from-outcome> ❶
    <to-view-id>/showproducts.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

```

<navigation-rule>
  <from-view-id>/showproducts.jsp</from-view-id>
  <navigation-case>
    <from-outcome>items.displayed</from-outcome> ②
    <to-view-id>/showitems.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

<navigation-rule>
  <from-view-id>/showitems.jsp</from-view-id>
  <navigation-case>
    <from-outcome>item.displayed</from-outcome> ③
    <to-view-id>/showitem.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

```

◀ De la page des produits, on se dirige vers la page des articles.

◀ À partir de la liste des articles, on navigue vers la page affichant le détail d'un produit.

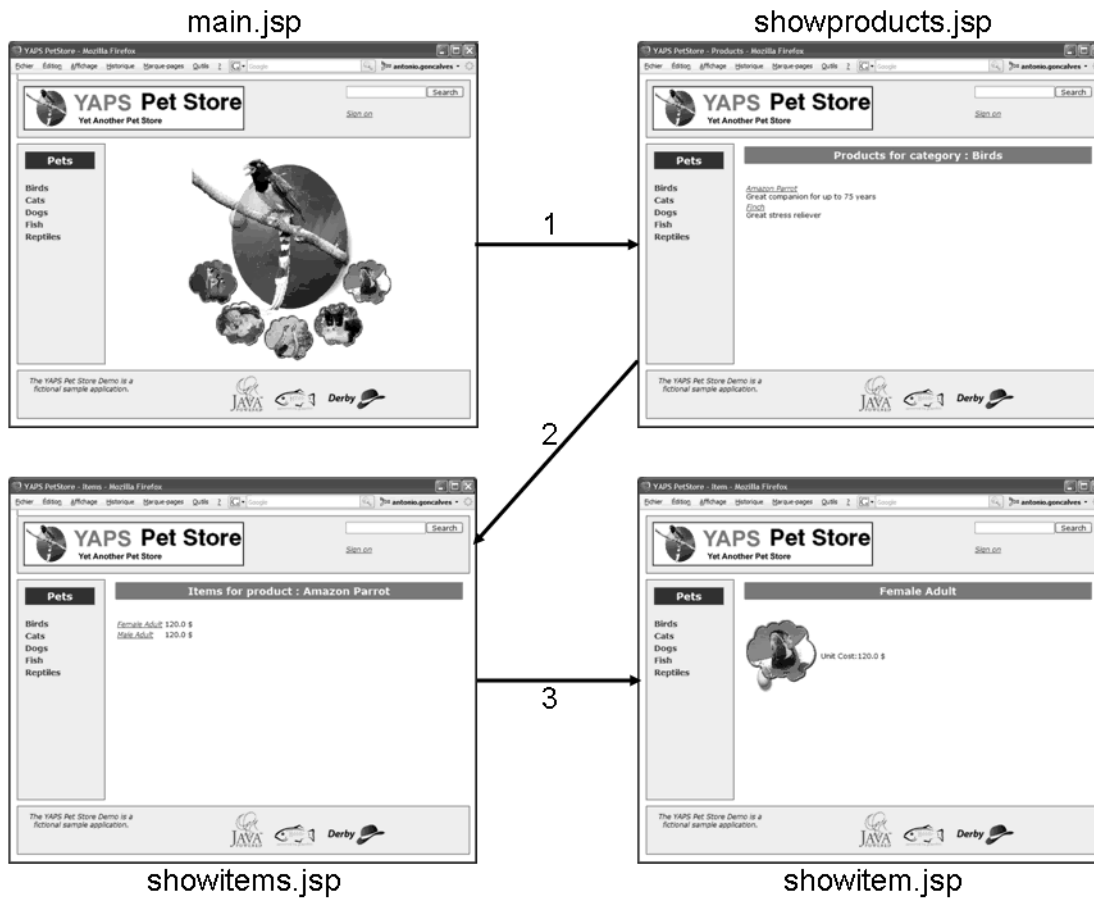


Figure 7-9 Navigation entre les pages du catalogue

La page d’affichage des produits

En cliquant sur une catégorie, la page `showproducts.jsp` affiche la liste de ses produits (figure 7–10).



Figure 7–10
La page `showproducts.jsp` affiche les produits d’une catégorie.

Extrait de la page `showproducts.jsp`

Cette page utilise les bibliothèques de balises JSF.

Titre de la page avec le nom de la catégorie.

Les éventuels messages d’erreurs sont affichés.

Itère la liste des produits

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<f:view>

  <h2>Products for category :
    <h:outputText value="#{catalog.category.name}"/>
  </h2>

  <h:messages layout="table" styleClass="error"/>

  <h:form>

    <h:dataTable value="#{catalog.products}" var="product">
      <h:column>
```

```

<h:commandLink action="#{catalog.doFindItems}">
  <h:outputText value="#{product.name}"/>
  <f:param name="productId" value="#{product.id}"/>
</h:commandLink>

<br/>
<h:outputText value="#{product.description}"/>
</h:column>
</h:dataTable>
</h:form>
</f:view>

```

◀ Affiche le nom du produit sous forme de lien. Lorsqu'on clique sur ce lien, la méthode `doFindItems` du managed bean est appelée en lui passant l'identifiant du produit comme paramètre (méthode `getParamId`).

REMARQUE Simplification du code

Pour en simplifier la lecture, la totalité du code des pages n'est pas détaillée. Retrouvez le code source complet de l'application à l'adresse :

► <http://www.antoniogoncalves.org>

La page d'affichage des articles

Un fois la liste des produits affichée, on peut cliquer sur un produit pour en connaître les articles (figure 7–11).



Figure 7–11

La page `showitems.jsp` affiche les articles d'un produit.

Titre de la page.	▶
Itère la liste des articles.	▶
Affiche le nom de l'article sous forme de lien. Cliquer sur le lien invoque la méthode doFindItem qui retournera l'article identifié par #{item.id}.	▶
Affiche le prix unitaire de l'article dans une colonne.	▶

WEB Packager des images

Chaque animal domestique est représenté par une image. Comme nous le verrons par la suite, celles-ci sont packagées avec l'application (JSP, CSS...) et accessibles sous le répertoire images.

Extrait de la page showitems.jsp

```
<h2>Items for product :
  <h:outputText value="#{catalog.product.name}"/>
</h2>

<h:messages layout="table" styleClass="error"/>
<h:form>

  <h:dataTable value="#{catalog.items}" var="item">

    <h:column>
      <h:commandLink action="#{catalog.doFindItem}">
        <h:outputText value="#{item.name}"/>
        <f:param name="itemId" value="#{item.id}"/>
      </h:commandLink>
    </h:column>

    <h:column>
      <h:outputText value="#{item.unitCost}"/> $
    </h:column>

  </h:dataTable>
</h:form>
```

La page de détail de l'article

Cette page permet d'afficher le détail de l'article sélectionné, c'est-à-dire son nom, son prix et une image (figure 7-12).



Figure 7-12
La page showitem.jsp
affiche le détail d'un article.

Extrait de la page showitem.jsp

```
<h2>
  <h:outputText value="#{catalog.item.name}"/>
</h2>
<h:messages layout="table" styleClass="error"/>
<h:form>
  <h:panelGrid columns="2">

    <h:column>
      <h:graphicImage url="images/#{catalog.item.imagePath}"/>
    </h:column>

    <h:column>
      <h:outputText value="Unit Cost:"/>
      <h:outputText value="#{catalog.item.unitCost}"/> $
    </h:column>
  </h:panelGrid>
</h:form>
```

- ◀ On dessine un tableau grâce à la balise panelGrid.
- ◀ Dans la première colonne, on affiche l'image de l'animal. Les images sont stockées dans le sous-répertoire images.
- ◀ Prix unitaire.

La gestion du compte par les clients

La gestion du compte client est détaillée dans les cas d'utilisation « Se connecter et se déconnecter », « Se créer un compte » et « Consulter et modifier son compte ». L'internaute peut se créer un profil en se choisissant un login et un mot de passe. Ensuite, en se connectant à l'application, il peut accéder à son profil et le mettre à jour.

Le managed bean AccountController

La gestion du compte client se fait via le managed bean AccountController. Celui-ci invoque l'EJB stateless CustomerBean qui manipule les données du client (entités Customer et Address). Le diagramme de classes suivant nous montre ces interactions (figure 7-13).

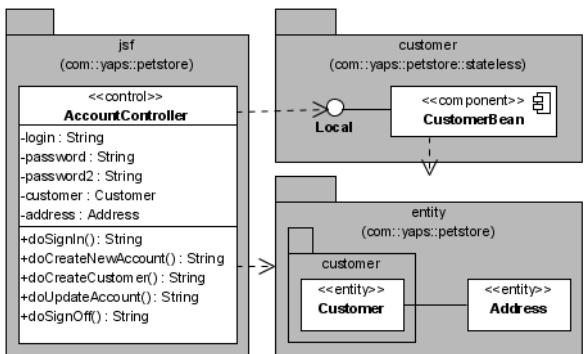


Figure 7-13
Diagramme du AccountController et de ses dépendances

L'injection de l'interface locale de l'EJB est réalisée à l'aide de l'annotation @EJB.

Attributs et entités manipulés par le managed bean.

À partir d'un login et d'un mot de passe, cette méthode permet d'identifier un client.

Cette méthode crée un client. L'internaute saisit ses informations dans un formulaire, ces données sont mappées sur les entités Customer et Address du managed bean, pour ensuite être passées à l'EJB.

Mise à jour du compte client.

Cette méthode déconnecte l'utilisateur du site. Pour cela, elle utilise le contexte JSF (FacesContext) pour récupérer la session HTTP et l'invalider.

Extrait du AccountController

```
public class AccountController

    @EJB
    private CustomerLocal customerBean;

    private String login;
    private String password;
    private String password2;
    private Customer customer = new Customer();
    private Address homeAddress = new Address();

    public String doSignIn() {
        customer = customerBean.authenticate(login, password);
        homeAddress = customer.getHomeAddress();
        return "customer.signed.in";
    }

    public String doCreateCustomer() {
        customer = customerBean.createCustomer(customer,
                                                homeAddress);
        homeAddress = customer.getHomeAddress();
        return "customer.created";
    }

    public String doUpdateAccount() {
        customer = customerBean.updateCustomer(customer,
                                                homeAddress);
        homeAddress = customer.getHomeAddress();
        return "account.updated";
    }

    public String doSignOff() {
        FacesContext fc = FacesContext.getCurrentInstance();
        HttpSession session = (HttpSession)
            fc.getExternalContext().getSession(false);
        session.invalidate();
        return "main";
    }

    // Getters & setters des attributs
}
```

Les pages web

La gestion du compte client peut se décliner en deux parties :

- connexion au site (signon.jsp) et création d'un compte client (createaccount.jsp);
- affichage des informations du client (showaccount.jsp) et possibilité de les mettre à jour (updateaccount.jsp).

La plupart de ces actions sont accessibles par des liens hypertextes contenus dans l'en-tête (header.jspf) de chaque page.

La navigation

Lorsque le client arrive sur le site, il peut soit visualiser le catalogue, soit se connecter. Pour cela, il peut cliquer sur le lien *Sign On* de l'en-tête. Il est alors redirigé sur une page lui demandant de saisir son login et son mot de passe (figure 7-14). Deux cas de figures sont possibles : soit l'utilisateur possède déjà un compte et le système le redirige vers la page d'accueil, soit il veut s'en créer un et le site lui propose de renseigner ses informations ❶.

Une fois connecté, le client peut à tout moment consulter son compte en cliquant sur le lien *Account* de l'en-tête. À partir de cette page, il peut également modifier ses informations ❷. Le lien *Sign Off* lui permet de se déconnecter.

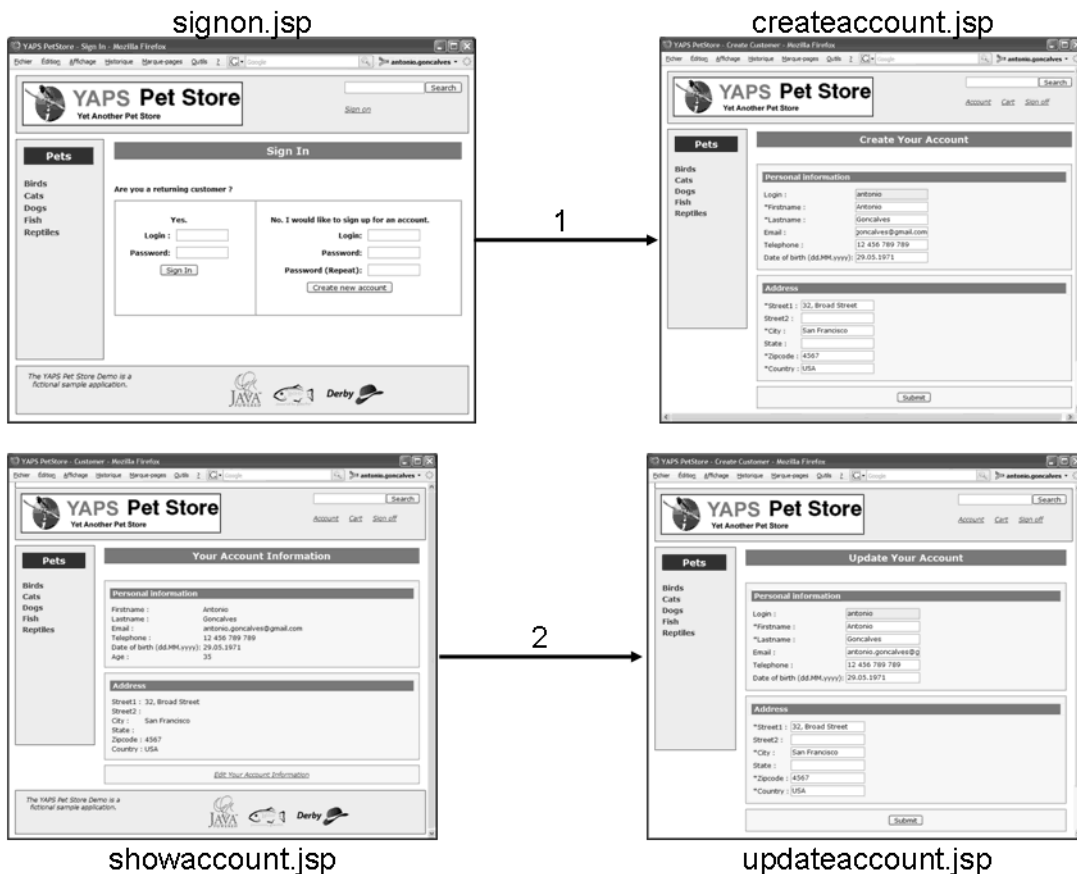


Figure 7-14 Navigation entre les pages du compte client

La navigation entre les pages du compte client est définie dans le fichier `faces-config.xml`.

Extrait du `faces-config.xml` concernant la navigation pour le compte client

À partir des liens placés dans l'en-tête, le client peut se connecter au site, se déconnecter ou afficher les informations de son compte.

►

```
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    <from-outcome>signon</from-outcome>
    <to-view-id>/signon.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>signoff</from-outcome>
    <to-view-id>/signoff.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>showaccount</from-outcome>
    <to-view-id>/showaccount.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Lorsque les données du compte client sont affichées, on peut les mettre à jour.

►

```
<navigation-rule>
  <from-view-id>/showaccount.jsp</from-view-id>
  <navigation-case>
    <from-outcome>updateaccount</from-outcome>
    <to-view-id>/updateaccount.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Une fois les données du compte mises à jour, on les réaffiche.

►

```
<navigation-rule>
  <from-view-id>/updateaccount.jsp</from-view-id>
  <navigation-case>
    <from-outcome>account.updated</from-outcome>
    <to-view-id>/showaccount.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Le client saisit son login et mot de passe. S'il est connu du système, il se connecte et la page principale est affichée.

►

```
<navigation-rule>
  <from-view-id>/signon.jsp</from-view-id>
  <navigation-case>
    <from-outcome>customer.signed.in</from-outcome>
    <to-view-id>/main.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Si, lors de la connexion, le client n'est pas reconnu par le système, on lui affiche un formulaire pour qu'il se crée un compte.

►

```
<navigation-rule>
  <from-view-id>/signon.jsp</from-view-id>
  <navigation-case>
    <from-outcome>create.a.new.account</from-outcome>
    <to-view-id>/createaccount.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

L'en-tête

Présente dans la partie haute de toutes les pages, l'en-tête propose des liens permettant aux clients de se connecter, de se déconnecter ou de consulter son compte.

Extrait du fragment de page header.jspf

```
<c:choose>
  <c:when test="${empty sessionScope.account}">
    <h:commandLink action="signon">
      <h:outputText value="Sign on"/>
    </h:commandLink>
  </c:when>
  <c:otherwise>
    <h:commandLink action="showaccount">
      <h:outputText value="Account"/>
    </h:commandLink>
    <h:commandLink action="#{account.doSignOff}">
      <h:outputText value="Sign off"/>
    </h:commandLink>
  </c:otherwise>
</c:choose>
```

- ◀ Balise Core JSP permettant de faire un test.
- ◀ Si le managed bean `account` ne se trouve pas dans la session HTTP, cela signifie que le client n'est pas connecté. On affiche alors le lien `Sign On` pour lui permettre de se connecter.
- ◀ Sinon, le client est déjà authentifié.
- ◀ Ce lien utilise la navigation statique pour accéder à la page du compte client.
- ◀ Lien permettant de se déconnecter.

La page de login

Cette page est découpée en deux parties (figure 7–15). Les clients qui ont déjà un compte saisissent leur login et mot de passe dans la partie gauche. Pour les autres, ils peuvent se choisir un login et saisir deux fois le même mot de passe. Ils seront ensuite redirigés vers un formulaire leur demandant de compléter les informations de leur compte client.

RETOUR D'EXPÉRIENCE S'authentifier à un site

Dans notre application, nous n'utilisons aucun mécanisme sophistiqué pour gérer l'authentification. En effet, on teste juste si le managed bean `account` se trouve dans la session HTTP ou non. Tout site qui se respecte doit avoir une politique de sécurité à toute épreuve. En standard, la spécification des servlets introduit un mécanisme d'authentification (`j_username`, `j_password`, `j_security_check`). S'il ne vous convient pas, vous pouvez aussi utiliser JAAS (*Java Authentication and Authorization Service*) ou un outil de Single Sign On (SSO).

- <https://opensso.dev.java.net/>
- <http://java.sun.com/products/jaas/>

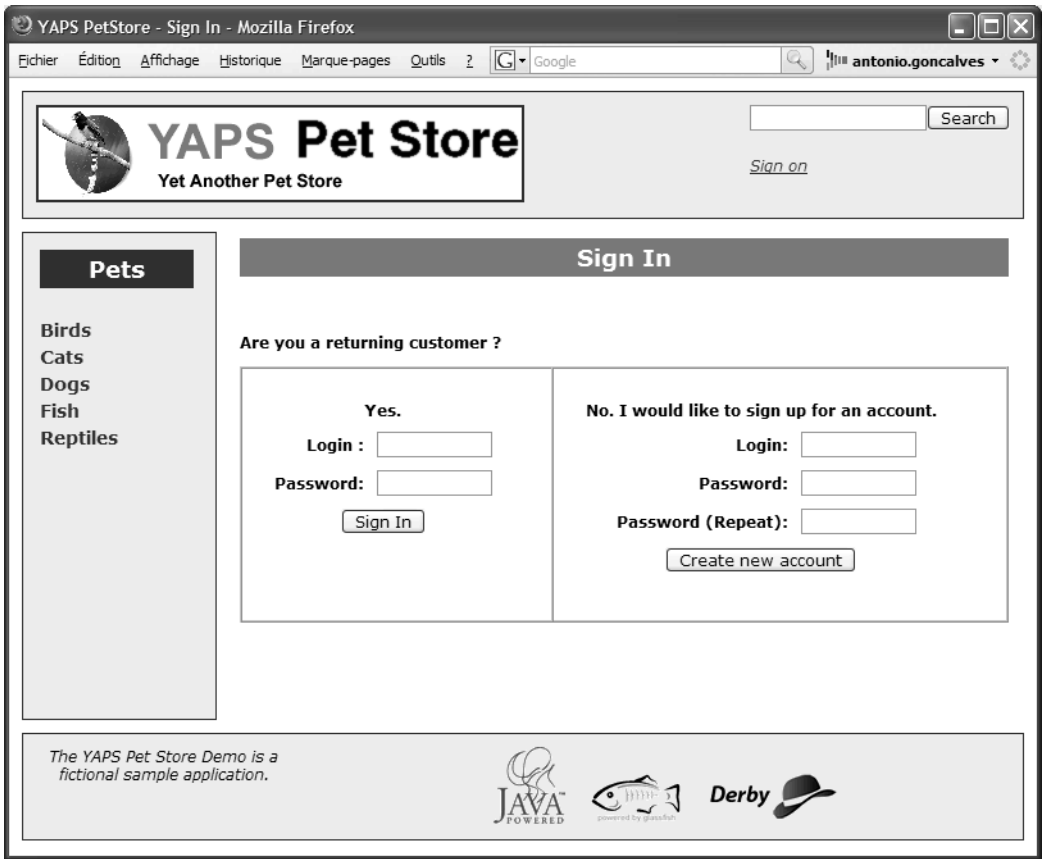


Figure 7–15
La page `signon.jsp`
permet l'authentification.

Extrait de la page `signon.jsp`

Les clients qui ont déjà un compte saisissent leur login et leur mot de passe avant de se connecter.

```
Are you a returning customer ?

Login :
    <h:inputText value="#{account.login}"
                 maxLength="10" size="12"/>
Password:
    <h:inputSecret value="#{account.password}"
                  maxLength="10" size="12"/>

    <h:commandButton value="Sign In" action="#{account.doSignIn}"
                     type="submit"/>

No. I would like to sign up for an account.

Login:
    <h:inputText value="#{account.customer.login}"
                 maxLength="10" size="12"/>
Password:
    <h:inputSecret value="#{account.customer.password}"
                  maxLength="10" size="12"/>
```

Les clients qui n'ont pas de compte, doivent saisir un login et deux fois le même mot de passe.

```

Password (Repeat):
    <h:inputSecret value="#{account.password2}"
                  maxLength="10" size="12"/>

<h:commandButton value="Create new account"
                  action="#{account.doCreateNewAccount}" type="submit"/>

```

Le formulaire de saisie

Cette page permet aux clients n'ayant pas de compte de s'en créer un en saisissant leurs informations (identité et adresse de domiciliation) (figure 7-16).

The screenshot shows a web browser window titled "YAPS PetStore - Create Customer - Mozilla Firefox". The address bar shows "Google" and the user is logged in as "antonio.goncalves". The page header includes the "YAPS Pet Store" logo and navigation links for "Account", "Cart", and "Sign off". A sidebar on the left lists "Pets" categories: Birds, Cats, Dogs, Fish, and Reptiles. The main content area is titled "Create Your Account" and contains two sections: "Personal information" and "Address". The "Personal information" section has input fields for Login (filled with "antonio"), *Firstname (filled with "Antonio"), *Lastname (filled with "Goncalves"), Email (filled with "goncalves@gmail.com"), Telephone (filled with "12 456 789 789"), and Date of birth (filled with "29.05.1971"). The "Address" section has input fields for *Street1 (filled with "32, Broad Street"), Street2, *City (filled with "San Francisco"), State, *Zipcode (filled with "4567"), and *Country (filled with "USA"). A "Submit" button is located at the bottom of the form.

Figure 7-16
createaccount.jsp affiche
un formulaire de création
de compte.

Identité du client.	▶
Tableau de deux colonnes.	▶
Login, nom, prénom, adresse e-mail et numéro de téléphone. Remarquez que les données du formulaire sont directement affectées à l'entity customer ({account.customer.login}...).	▶
La date de naissance est convertie au format jj/mm/aaaa.	▶
Adresse du client.	▶
Les informations de l'adresse sont affectées à l'entity Address.	▶
Une fois le formulaire saisi le client clique sur le bouton Submit pour se créer un compte.	▶

Extrait de la page createaccount.jsp

```
<h2>Create Your Account</h2>
<h:messages layout="table" styleClass="error"/>
<h:form>
  <h3>Personal information</h3>
  <h:panelGrid columns="2">
    <h:outputText value="Login :"/>
    <h:inputText value="#{account.customer.login}"
                  readonly="true"/>
    <h:outputText value="*Firstname :"/>
    <h:inputText value="#{account.customer.firstname}"/>
    <h:outputText value="*Lastname :"/>
    <h:inputText value="#{account.customer.lastname}"/>
    <h:outputText value="Email :"/>
    <h:inputText value="#{account.customer.email}"/>
    <h:outputText value="Telephone :"/>
    <h:inputText value="#{account.customer.telephone}"/>
    <h:outputText value="Date of birth (dd.MM.yyyy):"/>
    <h:inputText value="#{account.customer.dateOfBirth}"
                  <f:convertDateTime pattern="dd.MM.yyyy"/>
    </h:inputText>
  </h:panelGrid>
  <h3>Address</h3>
  <h:panelGrid columns="2">
    <h:outputText value="*Street1 :"/>
    <h:inputText value="#{account.homeAddress.street1}"/>
    <h:outputText value="Street2 :"/>
    <h:inputText value="#{account.homeAddress.street2}"/>
    <h:outputText value="*City :"/>
    <h:inputText value="#{account.homeAddress.city}"/>
    <h:outputText value="State :"/>
    <h:inputText value="#{account.homeAddress.state}"/>
    <h:outputText value="*Zipcode :"/>
    <h:inputText value="#{account.homeAddress.zipcode}"/>
    <h:outputText value="*Country :"/>
    <h:inputText value="#{account.homeAddress.country}"/>
  </h:panelGrid>
  <h:commandButton value="Submit"
                    action="#{account.doCreateCustomer}" type="submit"/>
</h:form>
```

L'affichage du compte client

Lorsque le client est connecté, le lien *Account* s'affiche dans l'en-tête. En cliquant sur ce lien, l'utilisateur est redirigé vers la page *showaccount.jsp* qui affiche les données du client en lecture seule (figure 7–17). Pour les mettre à jour, il suffit de cliquer sur le lien *Edit Your Account Information*.



Figure 7-17
La page showaccount.jsp affiche les informations du client.

Extrait de la page showaccount.jsp

```
<h2>Your Account Information</h2>
<h:messages layout="table" styleClass="error"/>
<h:form>
  <h3>Personal information</h3>
  <h:panelGrid columns="2">
    <h:outputText value="Firstname :"/>
    <h:outputText value="#{account.customer.firstname}"/>
    <h:outputText value="Lastname :"/>
    <h:outputText value="#{account.customer.lastname}"/>
    <h:outputText value="Email :"/>
    <h:outputText value="#{account.customer.email}"/>
    <h:outputText value="Telephone :"/>
    <h:outputText value="#{account.customer.telephone}"/>
```

◀ Section identité du client.

◀ Les balises outputText permettent d'afficher en lecture seule les attributs de l'entity Customer.

Remarquez l'accès à la variable `age` qui est calculée par l'entity à partir de la date de naissance.

Adresse de domiciliation.

Affiche les attributs de l'entity `Address`.

Le lien hypertexte utilise la navigation statique.

```
<h:outputText value="Date of birth (dd.MM.yyyy):"/>
<h:outputText value="#{account.customer.dateOfBirth}">
  <f:convertDateTime pattern="dd.MM.yyyy"/>
</h:outputText>

<h:outputText value="Age :"/>
<h:outputText value="#{account.customer.age}"/>

</h:panelGrid>

<h3>Address</h3>

<h:panelGrid columns="2">

  <h:outputText value="Street1 :"/>
  <h:outputText value="#{account.homeAddress.street1}"/>
  <h:outputText value="Street2 :"/>
  <h:outputText value="#{account.homeAddress.street2}"/>
  <h:outputText value="City :"/>
  <h:outputText value="#{account.homeAddress.city}"/>
  <h:outputText value="State :"/>
  <h:outputText value="#{account.homeAddress.state}"/>
  <h:outputText value="Zipcode :"/>
  <h:outputText value="#{account.homeAddress.zipcode}"/>
  <h:outputText value="Country :"/>
  <h:outputText value="#{account.homeAddress.country}"/>

</h:panelGrid>

<h:commandLink action="updateaccount">
  <h:outputText value="Edit Your Account Information"/>
</h:commandLink>

</h:form>
```

La mise à jour du compte client

Cette page affiche un formulaire pré-rempli des données du client et permet de les modifier (figure 7–18). Le code est presque identique à celui de la page `createaccount.jsp`, nous n'en détaillerons donc que quelques lignes.

Extrait de la page `updateaccount.jsp`

Identité du client.

Adresse de domiciliation du client.

Pour mettre à jour le compte client, le clic sur le bouton invoque la méthode `doUpdateAccount` du managed bean.

```
Personal information
(...)

Address
(...)

<h:commandButton value="Submit"
  action="#{account.doUpdateAccount}" type="submit"/>
```

YAPS Pet Store
Yet Another Pet Store

Account Cart Sign off

Pets

- Birds
- Cats
- Dogs
- Fish
- Reptiles

Update Your Account

Personal information

Login : antonio

*Firstname : Antonio

*Lastname : Goncalves

Email : antonio.goncalves@g

Telephone : 12 456 789 789

Date of birth (dd.MM.yyyy): 29.05.1971

Address

*Street1 : 32, Broad Street

Street2 :

*City : San Francisco

State :

*Zipcode : 4567

*Country : USA

Submit

Figure 7-18
updateaccount.jsp
permet de modifier le
compte client.

Gestion des erreurs

Les sources des managed beans et des pages ont été simplifiées pour en faciliter la lecture et la compréhension. Dans le code que nous avons vu précédemment, la gestion des erreurs a été supprimée.

Le comportement souhaité est qu'en cas d'exception, la page en cours se rafraîchisse en affichant le message d'erreur en rouge. Pour cela, nous avons besoin de deux éléments. Le premier est l'utilisation de la balise `<h:messages>` dans chaque page. En effet, cette balise permet d'afficher les erreurs détectées par le bean managé. Le deuxième est le code du managed bean qui permet de gérer cette fonctionnalité.

Le code ci-après nous montre comment un bean managé intercepte des exceptions et peut changer la navigation entre pages.

Cette variable contient la clé de navigation.	›
Bloc try.	›
Le traitement se fait sans exception, la clé de navigation est affectée.	›
En cas d'exception, on rajoute un message au contexte JSF.	›
Retourne la valeur null ou customer.created.	›
Rajoute un message dans le contexte de JSF.	›

Exemple de méthode traitant les erreurs

```
public String doCreateCustomer() {  
    String navigateTo = null; ❶  
    try {  
        customer = customerBean.createCustomer(customer, ❷  
                                                    homeAddress);  
        homeAddress = customer.getHomeAddress();  
        navigateTo = "customer.created"; ❸  
    } catch (Exception e) {  
        addMessage(e); ❹  
    }  
    return navigateTo; ❺  
}  
  
private void addMessage(String message) { ❻  
    FacesContext context = FacesContext.getCurrentInstance();  
    context.addMessage(null, new FacesMessage(  
        FacesMessage.SEVERITY_WARN, message, null));  
}
```

Chaque méthode déclare une variable de type String navigateTo ❶. Celle-ci contient la clé de navigation qui va être retournée par la méthode ❺ et qui sera interprétée par le contrôleur JSF pour afficher la page à suivre. Cette méthode effectue des traitements ❷, et si aucune exception n'est levée, la clé de navigation est affectée ❸. Par contre, en cas d'exception, la variable navigateTo reste égale à null, et un message d'erreur est rajouté au contexte JSF ❹. La valeur null est alors retournée ❺, ce qui entraîne le réaffichage de la page en cours avec un message d'erreur.

La méthode privée addMessage ❻ permet de rajouter des messages dans le contexte de JSF (FacesContext.addMessage()). La balise <h:messages> ❼ se charge alors d'afficher les messages contenus dans ce contexte.

Extrait de la page createaccount.jsp avec affichage des erreurs

```
<f:view>  
    (...)   
    <h2>Create Your Account</h2>  
    (...)   
    <h:messages/> ❼   
    (...)   
    <h3>Personal information</h3>  
    (...)   
</f:view>
```

ARCHITECTURE La validation des données

Dans la gestion des exceptions que nous décrivons précédemment, le système renvoie une exception (ValidationException) en signalant que le nom de famille est obligatoire. Si vous vous reportez au chapitre 4, *Objets persistants*, vous verrez que cette exception est levée dans une méthode annotée @PrePersist de l'entity Customer.

L'image ci-après nous montre le cas où le client veut valider son formulaire alors qu'il n'a pas saisi son nom, champ défini comme obligatoire (figure 7-19). La page se réaffiche avec le message Invalid last name.

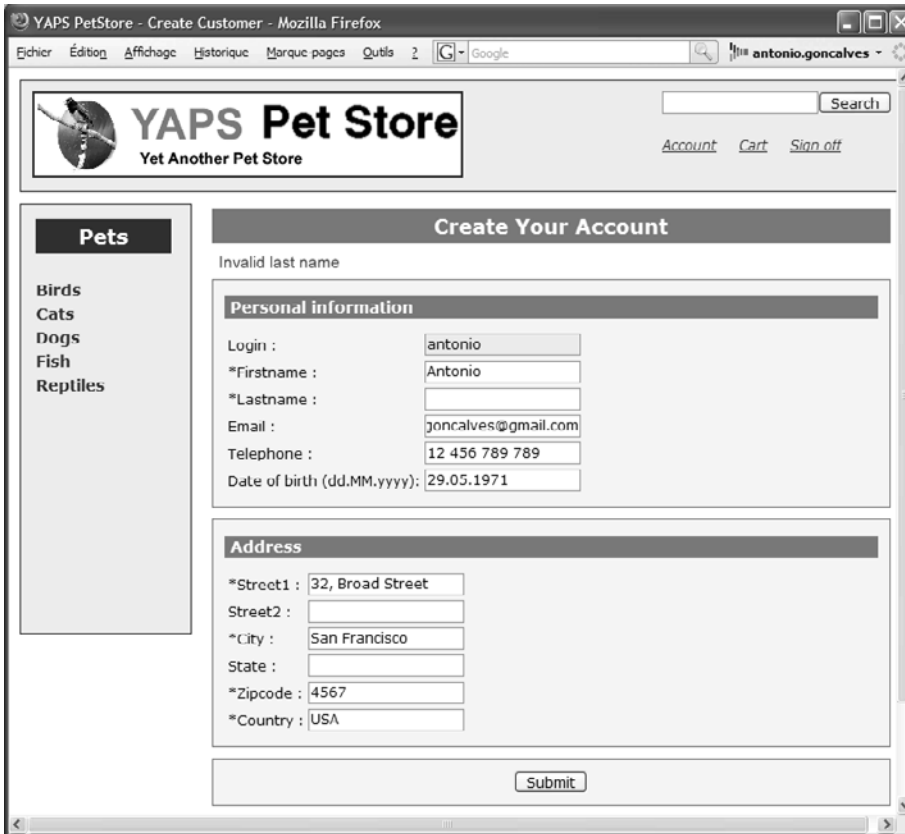


Figure 7-19
Affichage d'un message d'erreur

Paquetages et répertoires de l'interface web

Les managed beans AccountController et CatalogController sont placés dans le paquetage `com.yaps.petstore.jsf`.

L'application ne comporte plus seulement des classes Java, mais aussi des pages JSP, une feuille de style et des images. Tous ces éléments se situent dans le répertoire `resources`, au même niveau que le répertoire `src` qui contient les classes Java. Les JSP ainsi que la feuille de style (`petstore.css`) sont à la racine, les décorateurs (en-tête, bas de page et barre de navigation) sont dans le sous-répertoire `common`, et les images des animaux dans `images`.

Le descripteur de déploiement web (`web.xml`) et la configuration JSF (`faces-config.xml`) se trouvent dans le répertoire `WEB-INF`.

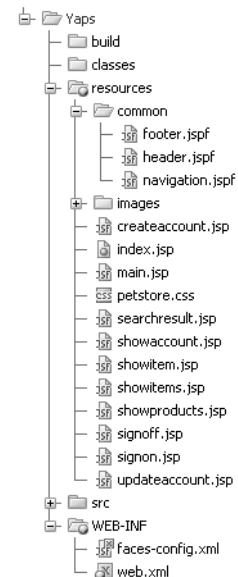


Figure 7-20 Éléments de l'application web

Architecture

Le diagramme de composants suivant nous montre comment s'insèrent les managed beans et l'application web (figure 7-21). La plomberie interne de JSF, comme le contrôleur, n'est pas détaillée dans ce diagramme. Le navigateur est donc représenté comme dialoguant directement avec les managed beans.

Le client Swing utilise les interfaces distantes des stateless beans, alors que les managed beans invoque les méthodes des interfaces locales. La couche des traitements manipule les données persistantes au travers d'entités. Quant à la couche de présentation, elle en affiche le contenu dans des pages JSP.

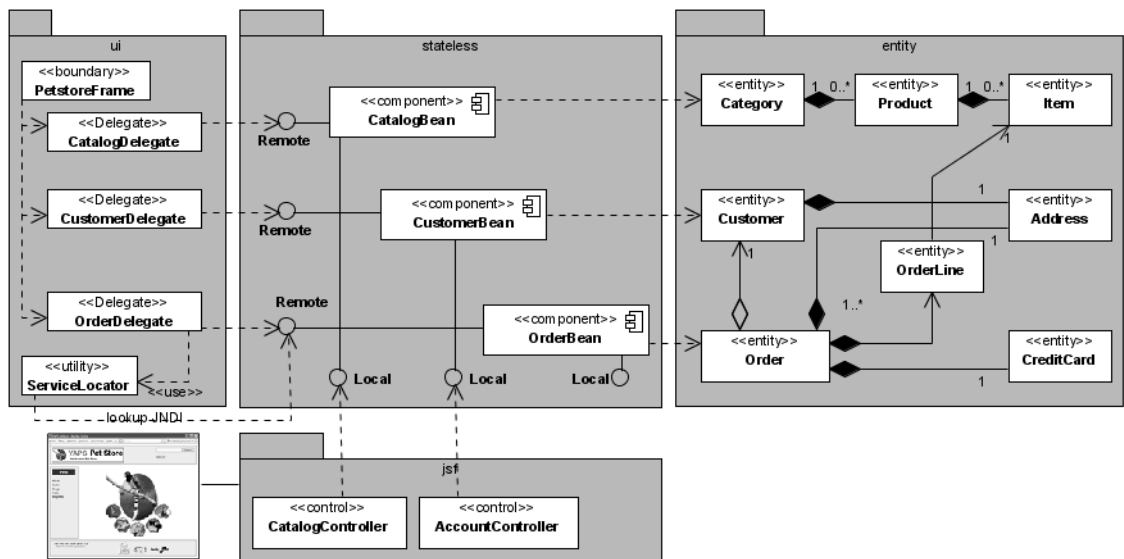


Figure 7-21 Le client web dans l'architecture globale de l'application

Exécuter l'application

Web Serveur et conteneur

Pour mettre à disposition des fichiers (pages HTML statiques, images, vidéo...) et gérer les requêtes HTTP, on doit utiliser un serveur web. Pour exécuter les servlets, les JSP, les taglibs JSTL ou JSF, il faut un conteneur web. GlassFish fournit toutes ces fonctionnalités.

Dans le précédent chapitre *Exécution de l'application*, nous avons déployé les entity et les stateless beans dans le conteneur d'EJB et exécuté l'interface Swing des employés. Désormais, il est également nécessaire de déployer l'application web et de l'utiliser à partir d'un navigateur.

La compilation des classes se fait toujours par l'exécution de la tâche Ant `yaps-compile`. Le packaging de l'application (`yaps-build`) est enrichi pour pouvoir rajouter une application web (on parle souvent de `webapp`) à l'archive entreprise (`.ear`).

Packager

Les applications web sont packagées dans des fichiers d'archive, appelés archives web, et qui portent une extension `.war` (Web ARchive). Ces archives permettent de regrouper tous les éléments d'une application web, que ce soit côté traitements (managed bean JSF, servlets, classes Java...) ou côté présentation (pages HTML, JSP, images, son, etc.). Cette extension du format JAR, spécialement dédiée aux applications web, a été introduite dans les spécifications 2.2 des servlets. C'est un format indépendant de toute plate-forme et exploitable par tous les conteneurs web qui respectent cette version de spécification.

La partie web de l'application YAPS Pet Store sera contenue dans le fichier `petstore.war`. À la racine de ce fichier, on trouve nos pages JSP, la feuille de style `petstore.css` et les images des animaux (sous-répertoire `images`). Les classes compilées des managed bean JSF doivent obligatoirement se trouver dans le sous-répertoire `WEB-INF/classes`. Ce répertoire est automatiquement ajouté par le conteneur au `CLASSPATH` lors du déploiement de l'application.

Pour que le conteneur web puisse interpréter le fichier `petstore.war` et déployer le contrôleur JSF (`FacesServlet`), il lui faut un descripteur. C'est le rôle du fichier `web.xml`, qui doit obligatoirement être présent dans le répertoire `WEB-INF`. On trouvera également à cet endroit le fichier de configuration `faces-config.xml`.

La tâche `yaps-build` packagera l'application entière dans le fichier `petstore.ear`. Celui-ci comporte :

- l'application web contenue dans le fichier `petstore.war` avec ses fichiers de description (`web.xml` et `faces-config.xml`) ;
- le fichier des EJB stateless (`stateless.jar`) ;
- les entités (`entity.jar`) et les classes utilitaires (`utility.jar`) accessibles depuis le sous-répertoire `lib`.

Déployer l'application et accéder au site

Pour déployer l'application, GlassFish doit être lancé (`ant -f admin.xml start-domain`) ainsi que la base de données Derby (`ant -f admin.xml start-db`). Utilisez ensuite la tâche `yaps-deploy` qui déploiera l'archive `petstore.ear` et initialisera la base de données. Vous n'avez plus qu'à prendre le navigateur de votre choix et vous rendre à l'adresse `http://localhost:8080/petstore` pour commencer à visualiser le catalogue et vous créer un compte (figure 7-23).

RAPPEL Les fichiers d'archive

Il existe plusieurs types de fichiers d'archive pour packager une application Java EE :

- les `.jar` (Java ARchive) pour les classes Java et les EJB ;
- les `.war` (Web ARchive) sont utilisés pour les applications web (servlet, JSP, JSF, images, HTML) ;
- les `.ear` (Enterprise ARchive) contiennent les fichiers JAR et WAR.

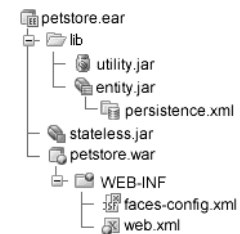


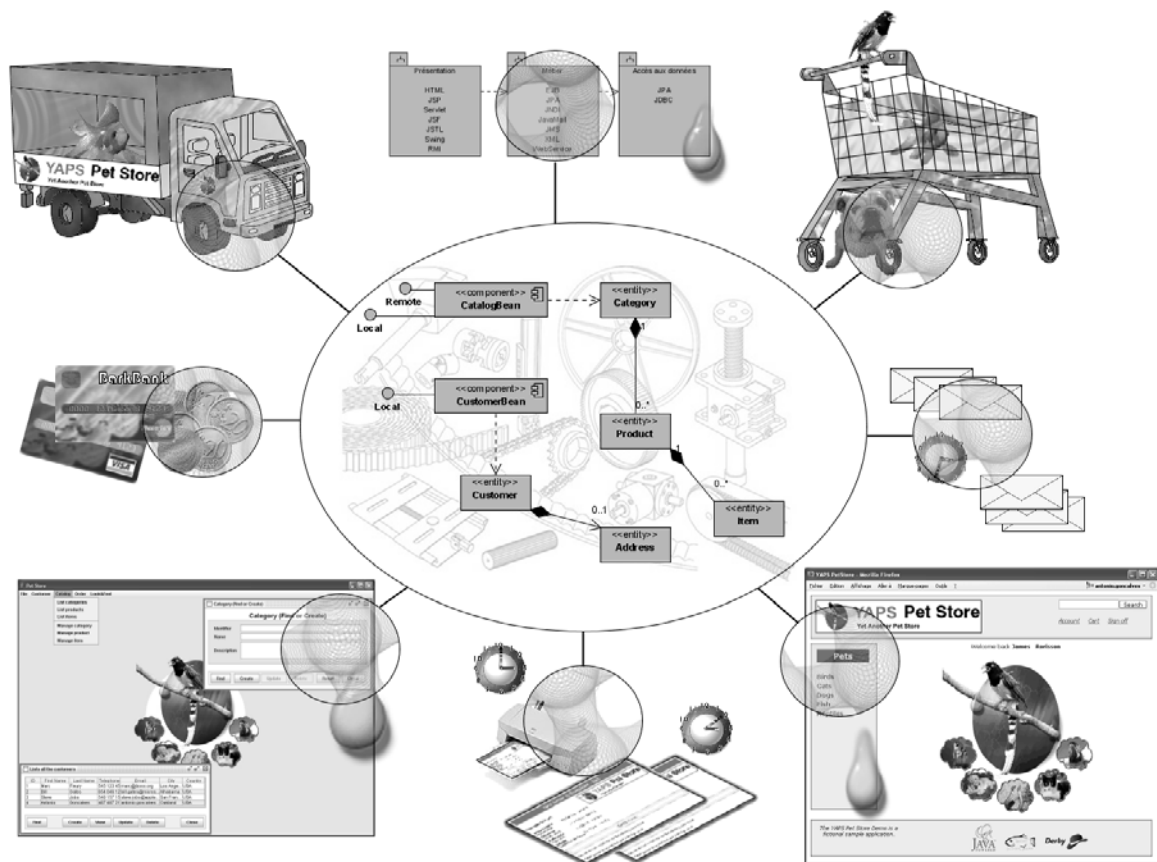
Figure 7-22 Contenu du fichier `petstore.ear`



Figure 7-23
Page d'accueil du site
YAPS Pet Store

En résumé

Pour accéder au système, les employés utilisent une interface Swing et les internautes un navigateur. Ce chapitre vous a présenté la panoplie de technologies utilisées pour développer une application web. Le modèle MVC de JSF nous permet de représenter graphiquement les données contenues dans les entités. Ceci est facilité par les managed beans qui délèguent les traitements aux EJB stateless. Le principe d'injection, qui facilite cette interaction, va être largement utilisé dans les chapitres suivants.



8

chapitre



Gestion du panier électronique

Les internautes peuvent désormais gérer leur compte client et visualiser le catalogue d'animaux. L'étape suivante consiste à autoriser ces clients à acheter des articles en ligne. Ce chapitre nous présente la gestion du Caddie électronique, ou plus exactement comment ajouter des articles dans un panier virtuel et en modifier la quantité. Ce panier est implémenté à l'aide d'un stateful session bean.

SOMMAIRE

- ▶ Acheter des animaux en ligne
- ▶ Gérer son panier électronique
- ▶ Conserver l'état dans une application web
- ▶ Stateful session bean
- ▶ Cycle de vie des stateful beans

MOTS-CLÉS

- ▶ Stateful bean
- ▶ HTTP Session

YAPS Pet Store propose un système de Caddie électronique pour les clients désirant acheter des animaux en ligne. Une fois authentifié, le client consulte le catalogue et peut ajouter des articles dans son panier virtuel. Il peut en rajouter autant qu'il le désire, en modifier la quantité, en supprimer, puis régler ses achats. Lorsque le client valide son panier, une commande est automatiquement créée. Ces interactions entre l'utilisateur et l'application web doivent être préservées. Si ce n'était pas le cas, l'application perdrait toute cohérence. Imaginez qu'à chaque ajout d'un produit, l'utilisateur doit s'identifier, pour ensuite s'apercevoir que tous les produits ajoutés précédemment aient disparus ! Il est donc nécessaire de préserver les interactions en mémoire.

La gestion de l'état dans une application web est difficile car le protocole sur lequel elle se base est intrinsèquement sans état. En effet, HTTP ne dispose pas de moyens pour se souvenir des interactions qui auraient pu se produire avec l'utilisateur. Le serveur web, avec l'aide du navigateur, doit donc maintenir les informations entre deux requêtes. C'est pour cette raison qu'il est nécessaire d'utiliser le concept de session (voir chapitre précédent, *Interface web*) si le serveur web souhaite garder un souvenir de l'état conversationnel.

En plus de la session HTTP, Java EE possède un autre moyen de stocker et de gérer cet état : les EJB stateful. L'application YAPS Pet Store utilise un stateful session bean pour implémenter son panier électronique.

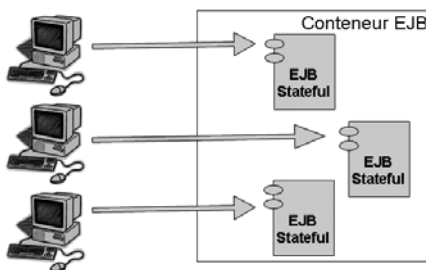


Figure 8-1
Un client pour un stateful bean

RETOUR D'EXPÉRIENCE **Stateful EJB vs HttpSession**

La plate-forme Java EE possède deux technologies lui permettant de stocker l'état d'une application web : les sessions HTTP et les stateful session beans. Il est parfois difficile de faire le choix entre les deux lorsqu'on a des données à stocker durant une session utilisateur. Voici une règle simple pour les différencier : la session HTTP est plutôt utilisée pour stocker la logique de présentation alors que le stateful stocke la logique métier. De plus, avec le stateful, cet état est accessible par les autres composants et non uniquement par le client web.

HTTP Session Object vs Stateful EJB

► <http://webservices.sys-con.com/read/42885.htm>

Stateful session bean

Par opposition aux stateless beans (sans état), les stateful session beans (avec état) associent les requêtes à un client spécifique, unissant client et bean dans une relation un à un. Ce type de composant peut aussi fournir un ensemble de méthodes métier, mais il dispose en plus de la possibilité de conserver des

données entre les différents appels d'un même client. Une instance est donc dédiée à un client qui sollicite ses services, et ce, tout au long du dialogue entre les deux protagonistes. Généralement, les méthodes proposées par un stateful bean permettent de consulter et de mettre à jour cet état.

Dans notre cas, le client veut pouvoir acheter plusieurs animaux domestiques. Il parcourt le catalogue, achète un caniche, puis quelques minutes plus tard, décide de le remplacer par un dalmatien, et achète en plus un poisson rouge. Tout cela peut prendre plusieurs minutes et à tout moment le client peut vouloir consulter le contenu de son panier électronique. Ce contenu (l'état de l'EJB) doit donc être accessible tout au long de cette conversation. Les statefuls maintiennent un état conversationnel, mais ils ne sont pas persistants (contrairement aux entités), c'est-à-dire que l'état ne survivrait pas à un arrêt du serveur, par exemple.

On peut voir un stateful bean comme une extension de l'application cliente. Les données conservées par le bean sont stockées dans des variables d'instances et conservées en mémoire, tout comme une application riche. Dans notre cas, la gestion du panier est propre à un client. Si au lieu d'utiliser un client léger nous utilisions une application Swing, la gestion du panier se ferait sur le poste client.

Exemple de stateful bean

Pour développer un stateful bean, les seuls objets nécessaires se résument à une classe contenant le code métier et, au minimum, une interface permettant les appels. Dans l'exemple ci-après, l'interface locale `ShoppingCartLocal` définit une méthode pour ajouter un nouvel article dans le panier. La classe `ShoppingCartBean` implémente cette interface.

Interface locale

```
@Local ❶
public interface ShoppingCartLocal {

    void addItem(Item item);
```

Classe du stateless bean

```
@Stateful ❷
public class ShoppingCartBean implements ShoppingCartLocal {

    private List<CartItem> cartItems; ❸

    public void addItem(Item item) {
        cartItems.add(new CartItem(item, 1)); ❹
    }
    // autres méthodes métiers
}
```

APPROFONDIR Stateful EJB

❷ Rima Patel Sriganesh, Gerald Brose, Micah Silverman, *Mastering Enterprise JavaBeans 3.0*, Wiley, 2006
 ▶ <http://www.jguru.com/faq/view.jsp?EID=917>

Ce code est très similaire à celui d'un stateless bean. En fait, les seules différences résident dans l'utilisation de l'annotation `@Stateful` ② au lieu de `@Stateless`, et dans le fait que les attributs ④ conservent leur état ⑤ entre les appels. Les interfaces restent identiques puisqu'elles sont annotées par `@Local` ① et doivent être implémentées par la classe ③.

EJB Annoter les interfaces

En EJB 3.0, il y a deux manières différentes d'utiliser les annotations `@Remote` et `@Local`.

- L'interface elle-même est annotée et la classe de l'EJB implémente cette interface (c'est la notation utilisée dans ce livre) :

@Local

```
public interface ShoppingCartLocal {
    void addItem(Item item);
}
```

@Stateful

```
public class ShoppingCartBean implements ShoppingCartLocal {
    (...)
    public void addItem(Item item) {
        (...)
    }
}
```

- L'interface n'est pas annotée, charge à la classe de l'EJB d'utiliser les annotations :

```
public interface ShoppingCartLocal {
    void addItem(Item item);
}
```

@Stateful

@Local(ShoppingCartLocal.class)

```
public class ShoppingCartBean implements ShoppingCartLocal {
    (...)
    public void addItem(Item item) {
        (...)
    }
}
```

Comment développer un stateful bean

Le développement d'un stateful bean est identique à celui d'un stateless (voir chapitre 5, *Traitements métier*) : il faut une classe ainsi qu'une ou deux interfaces.

Les interfaces

Un stateful bean peut avoir une interface distante et/ou locale. Son rôle étant de garder et de gérer un état, il a tendance à surtout être utilisé localement par une application web. Une application Swing, par exemple, n'a pas besoin de stateful bean pour gérer son état, elle peut le faire elle-même. On peut tout de même utiliser les deux types d'interfaces que l'on annotera avec `@Local` et/ou `@Remote`.

La classe de l'EJB

La classe du bean contient le code métier et implémente les interfaces. La nouveauté des stateless beans se caractérise par le fait que l'on peut déclarer des attributs en étant certain de leur contenu entre chaque appel.

La classe d'implémentation du bean

```
@Stateful ❶
public class ShoppingCartBean implements ShoppingCartLocal { ❷

    private List<CartItem> cartItems; ❸

    public void addItem(Item item) {
        cartItems.add(new CartItem(item, 1));
    }
    // autres méthodes métiers
}
```

Comme vous pouvez le voir, pour distinguer une simple classe Java d'un EJB stateful, il suffit d'utiliser l'annotation `@javax.ejb.Stateful` ❶. La classe implémente les méthodes de l'interface ❷ et maintient l'état de sa variable `cartItems` ❸.

Les attributs de l'annotation `@javax.ejb.Stateful` sont identiques à ceux de `@javax.ejb.Stateless`.

Code de l'annotation `@javax.ejb.Stateful`

```
package javax.ejb;

@Target(value = {TYPE}) @Retention(value = RUNTIME)
public @interface Stateful {

    String name() default "";

    String mappedName() default "";

    String description() default "";
}
```

RAPPEL Les interfaces

Reportez-vous au chapitre 5, *Traitements métier*, où les interfaces sont expliquées en détail.

SYNTAXE Classe EJB

La classe de l'EJB suit les règles de développement suivantes :

- elle doit être annotée par `@Stateful` ;
- elle doit être publique ;
- elle ne doit pas être finale ;
- elle ne doit pas être abstraite ;
- elle ne doit pas définir la méthode `finalize` ;
- elle doit avoir un constructeur par défaut ;
- elle doit implémenter les méthodes de ses interfaces.

- ◀ Cette annotation s'applique à une classe.
- ◀ Nom de l'EJB. Par défaut, le nom est celui de la classe.
- ◀ Cet attribut représente le nom donné à l'EJB à l'intérieur du conteneur. Il est spécifique à chaque serveur d'applications et peut donc ne pas être portable.
- ◀ Description du stateful session bean.

EJB Stateless vs stateful

- Stateless : pas de données internes ; peut être partagé entre plusieurs clients ; pas de passivation ; est stocké dans un pool.
- Stateful : conserve son état sur une suite d'appels ; dédié à un seul client ; pas de pool ; utilise la passivation et l'activation.

Le cycle de vie d'un stateful bean

La grande différence entre le cycle de vie d'un stateful et d'un stateless se résume au fait qu'il n'y a pas de pool. Pour les stateless beans, le serveur d'applications récupère n'importe quel EJB disponible dans un pool, puis lui transmet une requête à traiter. En effet, les stateless n'ayant pas d'état, ils peuvent exécuter un traitement indifféremment du client appelant. Le stateful, lui, appartient à un seul client, ce qui rend l'emploi du pool inutile. Par contre, lorsqu'il est inutilisé pendant un certain temps, le conteneur le supprime de la mémoire en le passivant (il sérialise alors ses données). Puis, lorsqu'un traitement doit être effectué, il le réactive.

Suivant l'implémentation du conteneur, le stateful bean a généralement une durée de vie (timeout) paramétrable. C'est-à-dire que si le client n'y accède pas durant une certaine période, l'EJB est supprimé du conteneur.

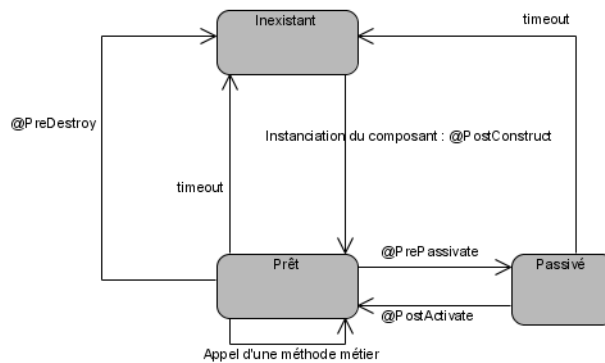


Figure 8–2
Cycle de vie d'un stateful bean

Activation et passivation

- Passivation : si l'EJB n'est pas utilisé, le conteneur enregistre l'état du bean en mémoire persistante (disque, base de données) et l'élimine de la mémoire.
- Activation : le conteneur réactive l'EJB et charge son état depuis la mémoire persistante. Il peut alors être réutilisé et invoqué par le client.

SYNTAXE Callback

Une méthode annotée par une annotation callback respecte les règles suivantes :

- elle peut être annotée par une ou plusieurs annotations de callback ;
- elle ne peut lancer que des `RuntimeException` ;
- elle peut être public, package, protected ou private.

L'état inexistant signifie que l'EJB n'a pas encore été instancié et n'existe pas en mémoire. Au premier appel d'une méthode, le cycle de vie s'initialise et reste dédié à un seul client. Lorsqu'il est à l'état prêt, le stateful bean garde son état conversationnel avec le client et répond à ses requêtes. Pendant sa durée de vie, il se peut qu'il ait des moments d'inactivité. Alors, pour libérer des ressources, le conteneur peut décider de le supprimer de la mémoire et de le passer pour une utilisation ultérieure (généralement, l'état de l'EJB est stocké sur disque). Il le réactivera (en le remettant en mémoire) à l'interception d'une nouvelle requête du client.

Les annotations de callback

Tout comme les entity et les stateless beans, le passage d'un état à l'autre peut être intercepté grâce à des annotations de callback. Il existe quatre annotations utilisables par les stateful beans.

- `@javax.annotation.PostConstruct`
- `@javax.annotation.PreDestroy`

- `@javax.ejb.PostActivate`
- `@javax.ejb.PrePassivate`

Après avoir instancié un stateful bean, le conteneur exécute les méthodes annotées par `@PostConstruct`. Dans le cas inverse, l'annotation `@PreDestroy` est appelée lorsque le conteneur supprime définitivement l'EJB. Deux autres annotations permettent d'intercepter le moment précédant une passivation (`@PrePassivate`) ou succédant à une activation (`@PostActivate`). Le développeur peut utiliser ces annotations pour libérer des ressources (par exemple, une connexion à une base ou à une file d'attente) avant que la passivation n'ait lieu, et les réinitialiser lors de l'activation de l'EJB.

Stateful bean utilisant des annotations de callback

```
@Stateful(name = "ShoppingCartSB")
public class ShoppingCartBean implements ShoppingCartLocal {
    private List<CartItem> cartItems;
    (...)

    @PostConstruct
    public void initialize() {
        cartItems = new ArrayList<CartItem>();
    }
}

@PreDestroy
public void clear() {
    cartItems = null;
}
}
```

- ◀ Le stateful bean utilise une liste d'objet.
- ◀ L'initialisation de la liste est faite après l'instanciation de l'EJB par le conteneur. L'annotation `@PostConstruct` intercepte cet événement et effectue un `new` sur la liste.
- ◀ À la destruction de l'EJB, le conteneur appelle cette méthode qui affecte la liste d'objet à `null`.

La gestion du Caddie de YAPS Pet Store

La gestion du panier est détaillée dans le cas d'utilisation « Acheter des articles » du premier chapitre. Un client visualise le catalogue et lorsqu'il est intéressé par un article, il l'ajoute dans son panier électronique. Le Caddie est utilisé uniquement par l'application web, il n'existe pas d'équivalent dans l'application Swing.

Le stateful bean

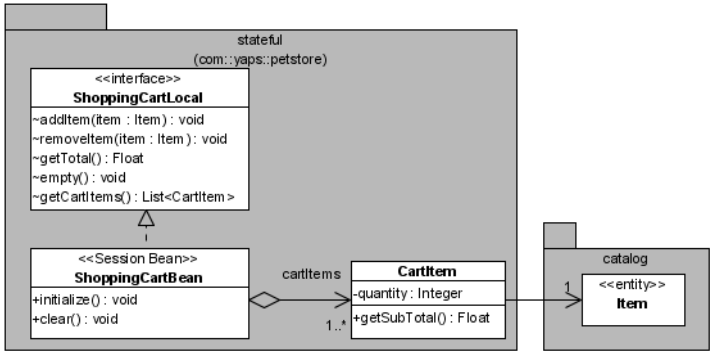
Le panier est représenté par un stateful session bean. Il possède un attribut de type `List` qui stocke les articles sélectionnés par le client, ainsi que leur quantité. Le composant `ShoppingCart` possède plusieurs méthodes permettant d'agir sur le contenu de cette liste :

- `addItem` : rajoute un article dans le Caddie.
- `removeItem` : supprime un article du Caddie.

- `getTotal` : retourne le montant total du Caddie.
- `empty` : vide le Caddie.
- `getCartItems` : retourne le contenu du panier, c'est-à-dire une liste de `CartItem`.

Le diagramme suivant nous montre les classes et interfaces impliquées dans la gestion du panier électronique (figure 8-3).

Figure 8-3
Diagramme de classes du ShoppingCart



ShoppingCartLocal

Les méthodes référencées dans l'interface locale sont utilisables par l'application web. Elles permettent au client d'ajouter des articles dans le panier électronique, de les supprimer, d'obtenir le prix total ou le contenu du Caddie. Ces actions se font uniquement au travers de l'interface locale, il n'y a donc pas d'interface distante.

Interface locale.

Méthodes manipulant le contenu du Caddie électronique.

```
@Local
public interface ShoppingCartLocal {
    void addItem(Item item);
    void removeItem(Item item);
    Float getTotal();
    void empty();
    List<CartItem> getCartItems() ;
}
```

ShoppingCartBean

La classe `ShoppingCartBean` implémente l'interface locale en ajoutant du code à toutes les méthodes. L'état du stateful bean (le contenu du panier) est stocké dans une liste d'objets `CartItem`.

Grâce à l'annotation, cette classe est identifiée comme étant un stateful bean qui se nomme `ShoppingCartSB`.

```
@Stateful(name = "ShoppingCartSB")
```

```

public class ShoppingCartBean implements ShoppingCartLocal {
    private List<CartItem> cartItems;

    @PostConstruct
    public void initialize() {
        cartItems = new ArrayList<CartItem>();
    }

    @PreDestroy
    public void clear() {
        cartItems = null;
    }

    public void addItem(Item item) {
        boolean itemFound = false;
        for (CartItem cartItem : cartItems) {
            if (cartItem.getItem().equals(item)) {
                cartItem.setQuantity(cartItem.getQuantity() + 1);
                itemFound = true;
            }
        }
        if (!itemFound)
            cartItems.add(new CartItem(item, 1));
    }

    public void removeItem(Item item) {
        for (CartItem cartItem : cartItems) {
            if (cartItem.getItem().equals(item)) {
                cartItems.remove(cartItem);
                return;
            }
        }
    }

    public Float getTotal() {
        if (cartItems == null || cartItems.isEmpty())
            return 0f;

        Float total = 0f;

        for (CartItem cartItem : cartItems) {
            total += (cartItem.getSubTotal());
        }
        return total;
    }

    public void empty() {
        cartItems.clear();
    }
}

```

- ◀ Le bean implémente l'interface locale.
- ◀ Liste des articles contenus dans le panier.
- ◀ Après initialisation de l'EJB par le conteneur, cette méthode est appelée. Elle permet d'instancier la liste de CartItem.
- ◀ Avant de détruire le stateful bean, le conteneur appelle cette méthode.
- ◀ Cette méthode ajoute un article dans le panier. Si l'article est déjà dans le panier, on en modifie la quantité.
- ◀ Cette méthode supprime un article du panier.
- ◀ Cette méthode additionne le prix de chaque article pour retourner le prix total du panier.
- ◀ Cette méthode vide le panier.

Simple objet Java sans annotation.
Référence l'entityItem ainsi que la quantité achetée.
Cette méthode retourne le sous-total (prix unitaire d'un article multiplié par sa quantité).

```
public class CartItem
{
    private Item item;
    private Integer quantity;

    public Float getSubTotal() {
        return item.getUnitCost() * quantity;
    }
    // getters & setters
}
```

CartItem

L'objet CartItem représente un élément du panier. En effet, il fait référence à l'entity Item (article) ainsi qu'à la quantité achetée. Le panier électronique est constitué d'une liste de CartItem.



Figure 8-4 Sous-paquetage stateful

Paquetages du stateful bean

L'interface, la classe du stateful bean ainsi que la classe CartItem se trouvent dans le paquetage com.yaps.petstore.stateful.

RETOUR D'EXPÉRIENCE Les noms des paquetages

Vous l'aurez remarqué, le nom de nos paquetages se rapporte aux technologies utilisées. Ainsi, le paquetage entity comporte tous les entités de l'application, stateless tous les stateless session beans, etc. Ce découpage en technologie est intéressant pour un apprentissage rapide : un simple nom évoque une API ou une spécification. Ce modèle a donc été utilisé pour, une fois de plus, simplifier la compréhension de Java EE 5 et vous aider à vous repérer rapidement dans le code.

En revanche, ce découpage est à proscrire dans un projet. Imaginez que vous laissiez les stateless beans pour utiliser de simples Pojo ou que les entités se transforment en classes Hibernate. Vous devriez alors changer le nom des paquetages.

Il est préférable de prendre des noms plus génériques tels que business (pour la couche de traitements) ou domain (pour les objets métiers). On peut aussi regrouper les classes par domaines plutôt que par couches. On aurait ainsi un paquetage catalogue qui comporterait les stateless beans, les entités, les exceptions, etc., liés au catalogue.

Le managed bean

Le managed bean ShoppingCartController fait le lien entre les pages et les appels aux composants métier. Il comporte toutes les méthodes liées à la gestion du panier (ajout et suppression d'un article) qu'il délègue au stateful bean, et invoque les stateless beans pour connaître les données du catalogue ou du client. Le diagramme de classes suivant nous montre ces interactions (figure 8-5).

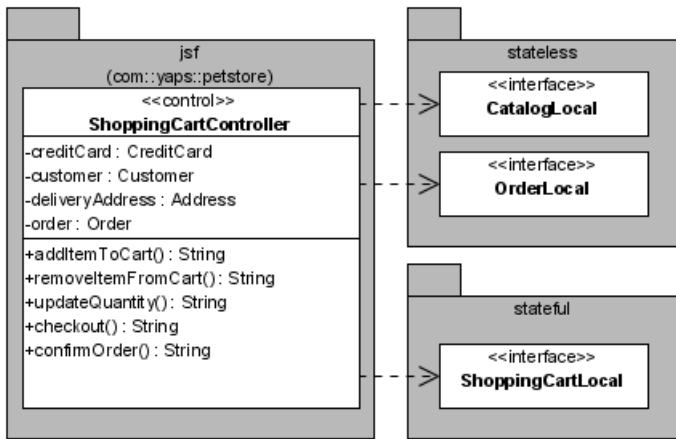


Figure 8–5
Diagramme du ShoppingCartController
et de ses dépendances

Extrait du ShoppingCartController

```

public class ShoppingCartController {
    @EJB
    private ShoppingCartLocal shoppingCartBean;
    @EJB
    private CatalogLocal catalogBean;
    @EJB
    private OrderLocal orderBean;
    private CreditCard creditCard = new CreditCard();
    private Order order;
    private Customer customer;
    private Address deliveryAddress;

    public String addItemToCart() {
        Item item = catalogBean.findItem(getParamId("itemId"));
        shoppingCartBean.addItem(item);
        return "item.added";
    }

    public String removeItemFromCart() {
        Item item = catalogBean.findItem(getParamId("itemId"));
        shoppingCartBean.removeItem(item);
        return null;
    }

    public String checkout() {
        return "cart.checked.out";
    }
}

```

Grâce à l'annotation @EJB, le conteneur injecte les références des interfaces locales des EJB stateless (CatalogBean et OrderBean) et de l'EJB stateful ShoppingCartBean.

Ces attributs sont initialisés par le faces-config.xml (voir plus bas). Ils correspondent aux informations du client qui est connecté.

Cette méthode permet d'ajouter un article au panier. La page web passe en paramètre l'identifiant de l'article (itemId) au bean qui utilise le stateless CatalogBean pour récupérer l'entity Item. Cet entity est rajouté dans le panier. La valeur de retour item.added est utilisée pour la navigation (voir le faces-config.xml).

Même type de traitement que la méthode ci-dessus. Notez la valeur de retour null. Celle-ci indique à JSF de réafficher la même page. En effet, lorsqu'on supprime un article du panier, une redirection est faite sur la page qui en affiche le contenu.

Lorsque le client a terminé ses achats des animaux et qu'il souhaite valider son panier, il clique sur le lien *Check Out*. Aucune action n'est effectuée. Seule une redirection vers la prochaine page est lancée.

Le client saisit son adresse de livraison, son moyen de paiement et valide ses achats. C'est à ce moment que le bon de commande est créé grâce au stateless `OrderBean` et que le panier est vidé.

Cette méthode permet d'afficher le prix total du panier.

Retourne le contenu du Caddie.

```
public String confirmOrder() {
    order = orderBean.createOrder(customer, deliveryAddress,
                                   creditCard, shoppingCartBean.getCartItems());
    shoppingCartBean.empty();
    return "order.confirmed";
}

public Float getTotal() {
    return shoppingCartBean.getTotal();
}

public List<CartItem> getCartItems() {
    return shoppingCartBean.getCartItems();
}
}
```

Les pages web

L'application web est enrichie de trois pages qui permettent :

- de consulter et de modifier le contenu du panier (`showcart.jsp`) ;
- de saisir l'adresse de livraison et le moyen de paiement (`confirmorder.jsp`) ;
- d'obtenir un récapitulatif des achats effectués et d'informer le client du numéro de bon de commande (`orderconfirmed.jsp`).

Pour utiliser son panier électronique, l'internaute doit au préalable être authentifié. Le menu *Add to cart* (ajouter au panier) s'affichera alors en face de chaque article. Un simple clic permet d'ajouter un article dans le panier. L'internaute peut alors en modifier la quantité, ou supprimer un article qu'il ne souhaiterait plus. Durant sa session, le client peut, à tout moment, visualiser le contenu de son panier en cliquant sur le menu *Cart* situé dans l'en-tête.

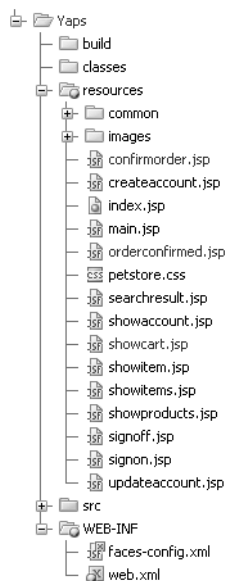


Figure 8-6 Éléments de l'application web

La navigation

À partir des pages affichant un article (`showitems.jsp` et `showitem.jsp`), il est possible de cliquer sur le lien *Add to cart* ❶ pour ajouter un article et consulter le contenu du panier (`showcart.jsp`). Cette page se réaffiche ❷ lorsque le client supprime un article du Caddie ou qu'il en modifie la quantité. Après validation de son panier (en cliquant sur le lien *Check Out*), le client est dirigé vers une page ❸ (`confirmorder.jsp`) lui demandant de saisir l'adresse de livraison ainsi que son numéro de carte bancaire. En confirmant, il est redirigé ❹ vers une page récapitulant le contenu de son panier et l'informant du numéro de bon de commande (`orderconfirmed.jsp`) (figure 8-7).

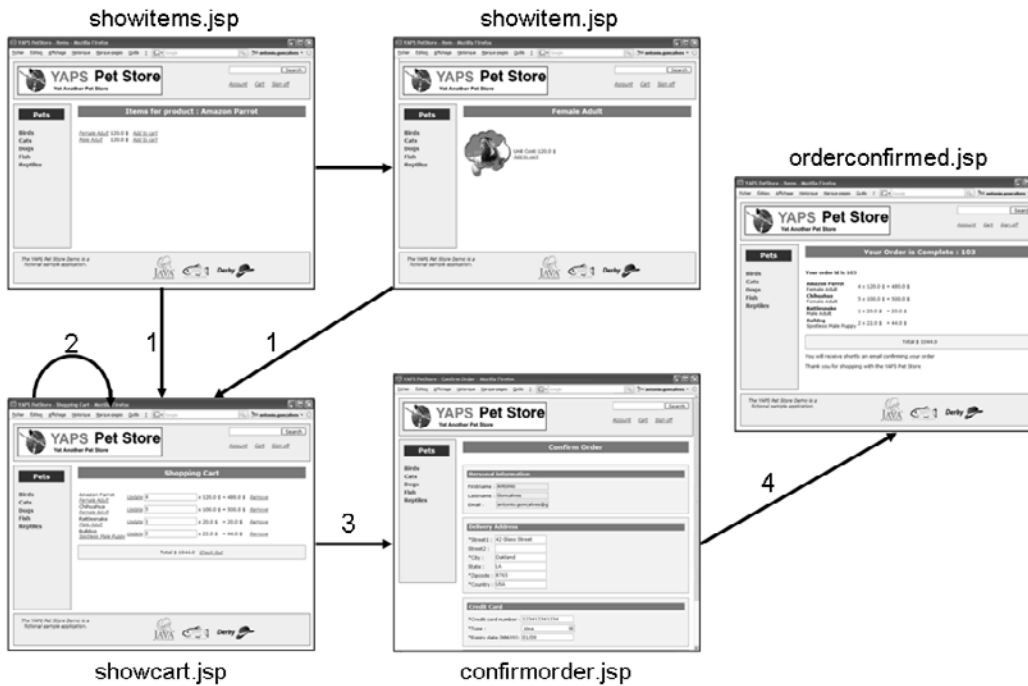


Figure 8–7
Navigation entre les pages
concernant le panier
électronique

Comme nous l'avons vu dans le précédent chapitre, *Interface web*, la navigation est définie dans le fichier `faces-config.xml`. Voici donc, au format compréhensible par JSF, l'enchaînement entre ces pages.

Extrait du `faces-config.xml` concernant la navigation

```
<navigation-rule>
  <from-view-id>/showitems.jsp</from-view-id>
  <navigation-case>
    <from-outcome>item.added</from-outcome>
    <to-view-id>/showcart.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

<navigation-rule>
  <from-view-id>/showitem.jsp</from-view-id>
  <navigation-case>
    <from-outcome>item.added</from-outcome>
    <to-view-id>/showcart.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

<navigation-rule>
  <from-view-id>/showcart.jsp</from-view-id>
  <navigation-case>
    <from-outcome>cart.checked.out</from-outcome>
    <to-view-id>/confirmorder.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

À partir de la page affichant la liste des articles, un clic sur **Add to cart** permet d'ajouter un article au panier et d'aller à la page `showcart.jsp` qui en affiche le contenu ❶.

À partir de la page affichant un article, un clic sur **Add to cart** permet d'ajouter cet article au panier et d'aller à la page `showcart.jsp` qui en affiche le contenu ❶.

Lorsqu'il consulte le contenu de son Caddie et qu'il veut le valider, le client clique sur **Check Out**. Il est alors redirigé vers la page `confirmorder.jsp` qui lui demande de saisir son adresse de livraison et son mode de paiement ❸.

Après confirmation de l'adresse et du mode de paiement, la page de récapitulatif `orderconfirmed` s'affiche 4.

```
<navigation-rule>
  <from-view-id>/confirmorder.jsp</from-view-id>
  <navigation-case>
    <from-outcome>order.confirmed</from-outcome>
    <to-view-id>/orderconfirmed.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Ajouter un article au Caddie

Lorsque le client est authentifié, et donc reconnu par le système, il peut acheter des articles. Pour cela, les pages `showitems.jsp` et `showitem.jsp` sont enrichies d'un lien *Add To Cart* (figure 8-8). Un simple clic sur ce lien et l'article en question est automatiquement ajouté dans le Caddie. Le client est alors redirigé vers la page `showcart.jsp` lui permettant de consulter l'état de son Caddie.



Figure 8–8

La page `showitem.jsp` avec le lien *Add To Cart*

Extrait de la page `showitem.jsp`

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
  prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsp/html" prefix="h" %>
<f:view>
<h:form>
```

```

<h:graphicImage url="images/#{catalog.item.imagePath}"/>
<h:outputText value="Unit Cost:"/>
<h:outputText value="#{catalog.item.unitCost}"/> $

<c:if test="${not empty sessionScope.account}">
  <br/>

  <h:commandLink action="#{cart.addItemToCart}">
    <h:outputText value="Add to cart"/>
    <f:param name="itemId" value="#{catalog.item.id}"/>
  </h:commandLink>
</c:if>
</h:form>
</f:view>

```

- ◀ La page affiche l'image de l'animal domestique ainsi que son prix.
- ◀ Si le client est connecté (c'est-à-dire que la session HTTP contient l'objet `account`)...
- ◀ ... alors le lien *Add To Cart* est affiché. Lorsque le client clique dessus, la méthode `addItemToCart` du managed bean est invoquée. Le numéro de l'article est passé en paramètre.

La page de contenu du Caddie

Cette page permet d'afficher et de modifier le contenu du panier électronique. Lorsque celui-ci est vide, un message en avertit le client. Sinon, la page `showcart.jsp` s'affiche avec pour chaque article son nom, sa description, sa quantité, son prix unitaire et le sous-total (prix unitaire × quantité). Le montant total du panier est également renseigné (figure 8–9).

PRÉCISION Le code des JSP

Pour que le code des JSP soit plus lisible, certaines balises comme `<h:column>` ou `<h:panelGrid>` ont été supprimées.

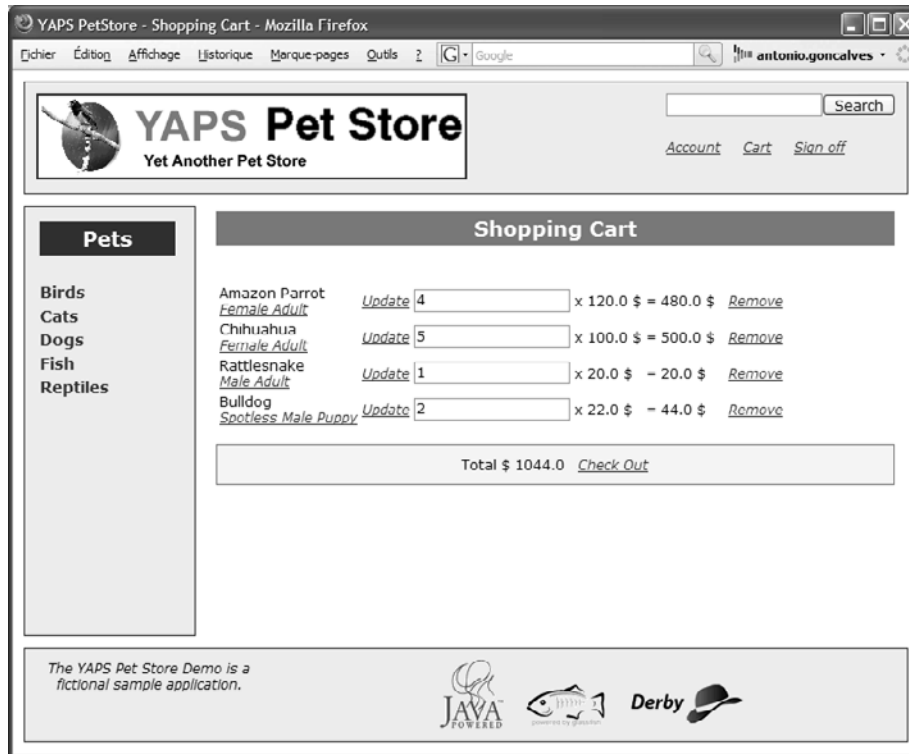


Figure 8–9

La page `showcart.jsp` affiche le contenu du panier.

Utilisation de l'Expression Language (EL) pour vérifier si le panier est vide.	▶
Si le Caddie n'est pas vide, on affiche son contenu.	▶
Permet d'itérer la liste des articles contenus dans le panier.	▶
Affiche le nom du produit.	▶
Affiche un lien vers l'article. Notez que l'identifiant est passé en paramètre au managed bean (voir l'appel à la méthode <code>getParamId()</code> pour obtenir la valeur de ce paramètre).	▶
Affiche la ligne « 4 × 120 \$ = 480 \$ », par exemple.	▶
Lien pour supprimer l'article du Caddie.	▶
Affiche le prix total du panier.	▶

Extrait de la page `showcart.jsp`

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>

<f:view>
<c:choose>

  <c:when test="${empty sessionScope.cart.cartItems}">
    The Shopping Cart is empty
  </c:when>

  <c:otherwise>
    <h:form>

      <h:dataTable value="#{cart.cartItems}" var="cartItem">

        <h:outputText value="#{cartItem.item.product.name}"/>

        <h:commandLink action="#{catalog.doFindItem}">
          <h:outputText value="#{cartItem.item.name}"/>
          <f:param name="itemId" value="#{cartItem.item.id}"/>
        </h:commandLink>

        <h:commandLink action="#{cart.updateQuantity}">
          <h:outputText value="Update"/>
          <f:param name="itemId" value="#{cartItem.item.id}"/>
        </h:commandLink>

        <h:inputText value="#{cartItem.quantity}"/>
        x <h:outputText value="#{cartItem.item.unitCost}"/> $
        = <h:outputText value="#{cartItem.subTotal}"/> $

        <h:commandLink action="#{cart.removeItemFromCart}">
          <h:outputText value="Remove"/>
          <f:param name="itemId" value="#{cartItem.item.id}"/>
        </h:commandLink>
      </h:dataTable>

      Total $ <h:outputText value="#{cart.total}"/>
    </c:otherwise>
  </c:choose>
</f:view>
```

La page de saisie des données de livraison et de paiement

Lorsque le client valide le contenu du Caddie, il est redirigé vers la page `confimorder.jsp` (figure 8–10). Celle-ci est un formulaire composé de trois parties :

- le nom, prénom et adresse e-mail du client sont affichés à titre informatif en lecture seule ;
- une zone permettant de saisir l'adresse de livraison. Par défaut, le système affiche l'adresse de domiciliation du client ;
- une zone pour le mode de paiement, c'est-à-dire numéro de carte bancaire, type et date d'expiration.

Figure 8–10 La page `confirmorder.jsp` permet la saisie des informations de livraison.

Extrait de la page `confirmorder.jsp`

```
<%@ taglib uri="http://java.sun.com/jsp/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsp/html" prefix="h" %>
<f:view>
<h:form>
  <h3>Personal information</h3>
  <h:outputText value="Firstname :"/>
  <h:inputText value="#{cart.customer.firstname}"
    readonly="true"/>
  <h:outputText value="Lastname :"/>
  <h:inputText value="#{cart.customer.lastname}"
    readonly="true"/>
  <h:outputText value="Email :"/>
  <h:inputText value="#{cart.customer.email}"
    readonly="true"/>
```

◀ Affiche le nom, prénom et adresse e-mail du client en lecture seule.

Le formulaire de saisie de l'adresse de livraison est pré-initialisé par l'adresse de domiciliation du client (voir fichier faces-config.xml).

Formulaire de saisie du moyen de paiement.

Le type de carte de crédit est présenté dans une combo-box.

```
<h3>Delivery Address</h3>
<h:outputText value="*Street1 :"/>
<h:inputText value="#{cart.deliveryAddress.street1}"/>
<h:outputText value="Street2 :"/>
<h:inputText value="#{cart.deliveryAddress.street2}"/>
<h:outputText value="*City :"/>
<h:inputText value="#{cart.deliveryAddress.city}"/>
<h:outputText value="State :"/>
<h:inputText value="#{cart.deliveryAddress.state}"/>
<h:outputText value="*Zipcode :"/>
<h:inputText value="#{cart.deliveryAddress.zipcode}"/>
<h:outputText value="*Country :"/>
<h:inputText value="#{cart.deliveryAddress.country}"/>

<h3>Credit Card</h3>
<h:outputText value="*Credit card number :"/>
<h:inputText value="#{cart.creditCard.creditCardNumber}"/>
<h:outputText value="*Type :"/>
<h:selectOneMenu value="#{cart.creditCard.creditCardType}">
  <f:selectItem itemValue="visa" itemLabel="Visa"/>
  <f:selectItem itemValue="visa_gold" itemLabel="Visa Gold"/>
  <f:selectItem itemValue="master" itemLabel="Master Card"/>
  <f:selectItem itemValue="american"
    itemLabel="American Express"/>
</h:selectOneMenu>
<h:outputText value="*Expiry date (MM/YY):"/>
<h:inputText value="#{cart.creditCard.creditCardExpDate}"/>
</h:form>
</f:view>n
```

Lorsque cette page s'affiche, elle est pré-initialisée par les données du client ainsi que son adresse de domiciliation. Ces informations sont gérées et manipulées par le managed bean AccountController et non ShoppingCartController. JSF a la possibilité d'initialiser les attributs d'un managed bean à partir d'attributs d'un autre managed bean. Cette action se fait de manière déclarative dans le fichier faces-config.xml. Dans notre cas, on veut initialiser l'attribut customer et deliveryAddress du ShoppingCartController avec les valeurs des attributs customer et homeAddress du AccountController.

Extrait du fichier faces-config.xml

Le managed bean AccountController a comme alias account.

Le managed bean ShoppingCartController a comme alias cart.

```
<managed-bean>
  <managed-bean-name>account</managed-bean-name>
  <managed-bean-class>
    com.yaps.petstore.jsf.AccountController
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>cart</managed-bean-name>
  <managed-bean-class>
    com.yaps.petstore.jsf.ShoppingCartController
```

```
</managed-bean-class>
<managed-bean-scope>session</managed-bean-scope>

<managed-property>
  <property-name>customer</property-name>
  <value>#{account.customer}</value>
</managed-property>

<managed-property>
  <property-name>deliveryAddress</property-name>
  <value>#{account.homeAddress}</value>
</managed-property>
</managed-bean>
```

- ◀ L'attribut `customer` du managed bean `ShoppingCartController` est initialisé avec `#{account.customer}` (c'est-à-dire l'attribut `customer` du managed bean ayant comme alias `account`).
- ◀ L'adresse de livraison est initialisée avec l'adresse de domiciliation.

La page récapitulative

L'adresse et le mode de paiement saisis, le client est redirigé vers une page récapitulant ses achats (figure 8–11). Cette page l'informe du numéro de bon de commande ainsi que de la réception imminente d'un e-mail (voir chapitre 10, *Traitements asynchrones*).

TÉLÉCHARGER Le code de l'application

Retrouvez le code de l'application YAPS Pet Store sur le site :

▶ <http://www.antoniogoncalves.org>

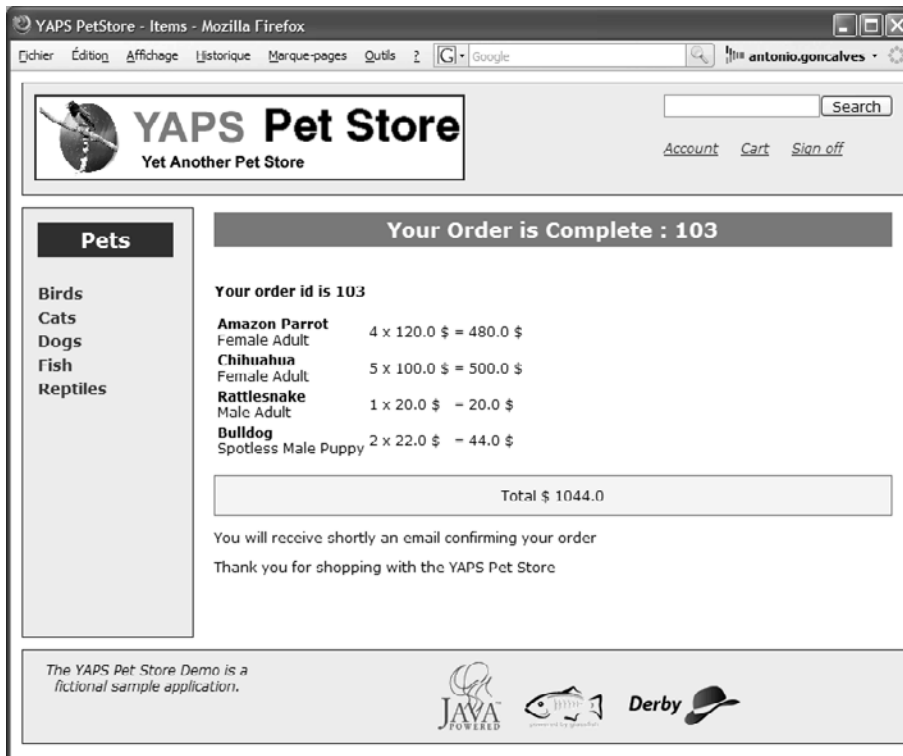


Figure 8–11
La page `orderconfirmed.jsp` affiche le récapitulatif.

Le code de cette page est très similaire à celui de `showcart.jsp`. Il ne sera donc pas décrit ici.

Architecture

L'architecture se voit enrichie d'un nouveau managed bean (ShoppingCartController). Celui-ci a pour tâche d'invoquer les méthodes du stateful bean ShoppingCartBean lorsque l'utilisateur souhaite accéder à son panier électronique. Le stateful bean n'expose qu'une interface locale et utilise une liste de Pojo (CartItem) pour stocker le contenu du Caddie.

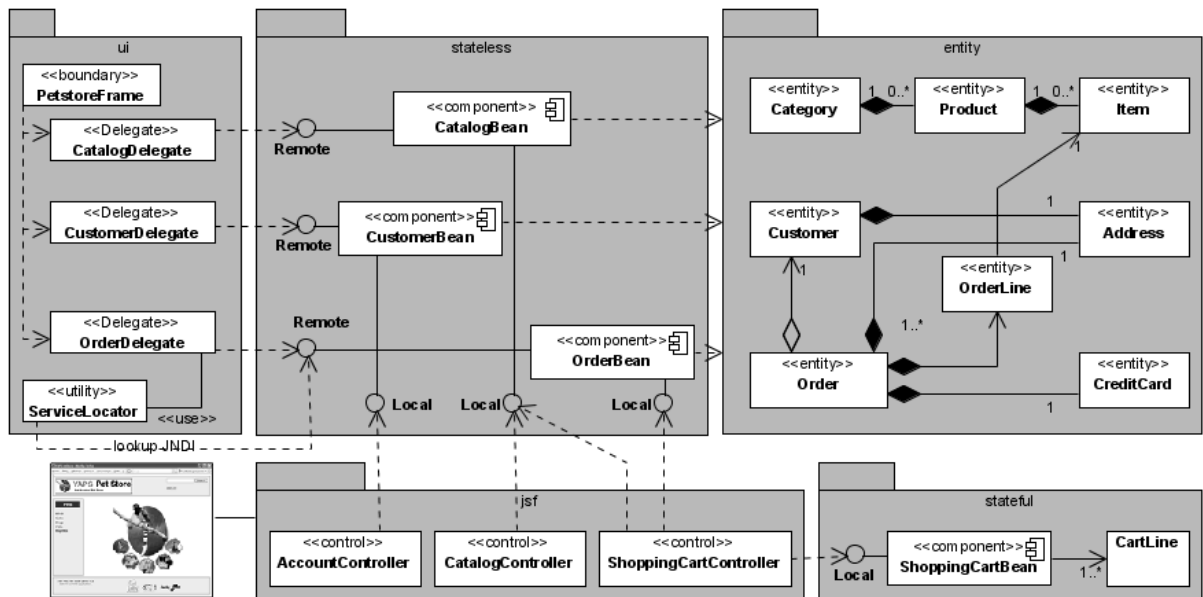


Figure 8-12 Stateful bean s'inscrivant dans l'architecture globale de l'application

Exécuter l'application

Pour exécuter l'application, il suffit d'utiliser les mêmes tâches Ant `yaps-clean`, `yaps-build` et `yaps-deploy` qui compileront, packageront les classes et déploieront les fichiers archives dans GlassFish. Le stateful session bean `ShoppingCartBean` est déployé dans un fichier JAR autonome : `stateful.jar`.

L'application à déployer sur le serveur est packagée dans le fichier `petstore.ear` qui contient :

- les EJB stateless (`stateless.jar`) ;
- l'EJB stateful (`stateful.jar`) ;

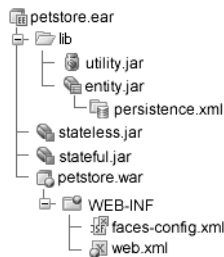


Figure 8-13 Contenu du fichier `petstore.ear`

- les entités (`entity.jar`) et les classes utilitaires (`utility.jar`) accessibles depuis le sous-répertoire `lib` ;
- l'application web contenue dans le fichier `petstore.war` avec ses descripteurs de déploiement (`web.xml` et `faces-config.xml`).

Vous pouvez maintenant utiliser l'application web pour acheter des animaux domestiques, et l'interface Swing pour consulter les bons de commande qui sont créés (figure 8-14).

REMARQUE Stateful dans l'arbre JNDI

Après le déploiement, si vous consultez l'arbre JNDI à partir de la console GlassFish, vous noterez que le stateful bean n'y apparaît pas. En effet, celui-ci n'ayant qu'une interface locale, GlassFish ne l'ajoute pas dans son annuaire. Le stateful bean ne peut donc pas être recherché (lookup) par un client distant.

The screenshot shows the 'Pet Store' application window. It has a menu bar with 'File', 'Customer', 'Catalog', 'Order', and 'Look&Feel'. Below the menu is a toolbar with a checkbox 'Lists all the orders' and some window controls. The main area contains a table of orders:

ID	Date	Customer Name	Number of it..	Total
1	05/05/2006	Jobs Steve	4	404.0
2	31/10/2006	Fleury Marc	2	96.0
3	28/08/2006	Gates Bill	3	342.0
1111	14/11/2006	Jobs Steve	1	120.0
103	19/12/2006	Jobs Steve	1	150.0

Below the table is a window titled 'Order - 1 (Read only)'. It contains a form for order details:

Order - 1 (Read only)

Identifier: 1
Order date: 05/05/2006

Customer

First Name: Steve
Last Name: Jobs
Date of birth: 30/11/1964

Address

Street 1: 154 Star Boulevard
Street 2:
City: San Francisco
State: WC
Zipcode: 5455
Country: USA

Delivery Address

Street 1: 154 Star Boulevard
Street 2:
City: San Francisco
State: WC
Zipcode: 5455
Country: USA

Total 404.0

Credit card

Number: 1234 5648 7890
Type: Visa
Expiry date: 09/09

At the bottom of the window are buttons: Find, Create, Update, Delete, Reset, and Close.

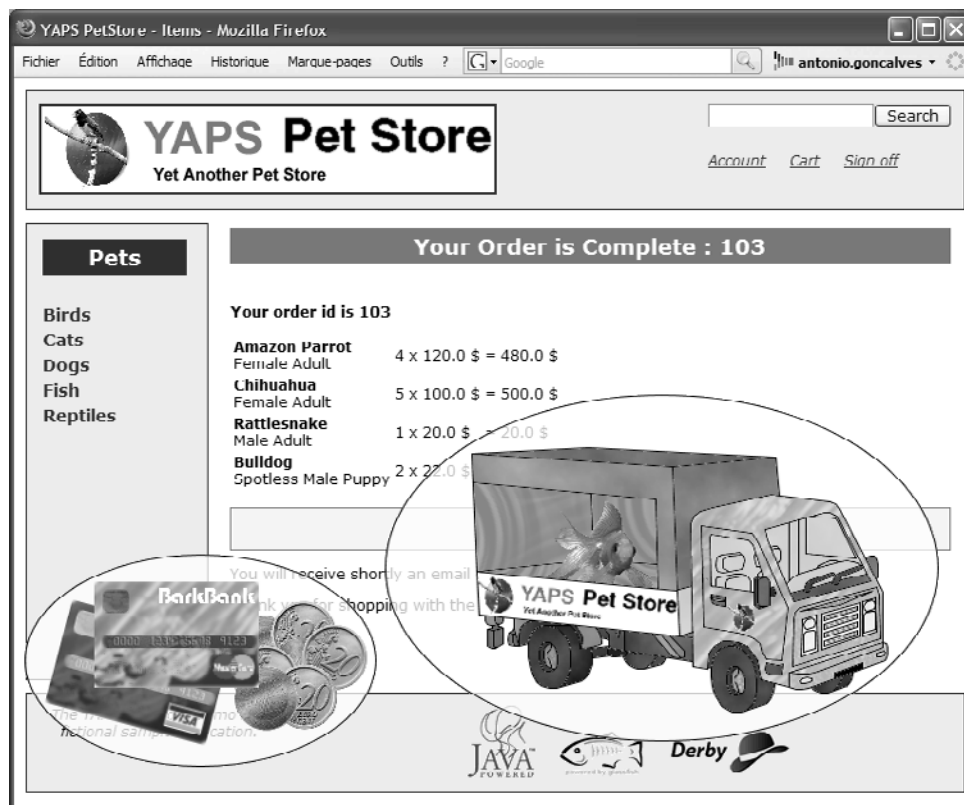
Figure 8-14 Affichage du détail d'un bon de commande

En résumé

Dans ce chapitre, nous avons enrichi l'application web existante en y ajoutant un panier électronique. Cette nouvelle fonctionnalité permet aux clients d'acheter en ligne des animaux domestiques. Pour cela, nous avons utilisé les stateful session beans de Java EE 5. Les managed beans de JSF font l'intermédiaire entre la représentation graphique du panier (JSP) et son contenu (EJB stateful).

9

chapitre



Échanges B2B

Lorsqu'un client valide ses achats, les données de sa carte bancaire doivent être vérifiées. Une fois le bon de commande créé, le transporteur doit être averti pour livrer les animaux à leurs propriétaires. Ces échanges avec les sociétés partenaires BarkBank et PetEx se font au travers de services web. Ce chapitre nous présente ces technologies, puis un exemple de développement et de test de services web.

SOMMAIRE

- Systèmes externes et interopérabilité
- Validation de la carte bancaire
- Avertir le transporteur
- Les services web
- Technologies autour des services web
- Génération des artefacts
- Déployer et tester un service web

MOTS-CLÉS

- Services web
- Soap
- WSDL
- JAX-WS
- JAXB

/// B2A, B2B, B2C, C2C

Ces sigles distinguent le commerce inter-entreprises (*Business to Business* ou B2B) du commerce avec les particuliers (*Business to Consumer* ou B2C). Il y a aussi les échanges entreprises-administrations (B2A ou *Business to Administration*) et ceux entre particuliers comme les enchères, petites annonces, etc. (*Consumer to Consumer* ou C2C).

/// URL, URI, URN, URC

URL (*Uniform Resource Locator*) est une chaîne de caractères servant à localiser des ressources consultables à l'aide d'un navigateur. Les URL sont un sous-ensemble d'URI (*Unified Resource Identifier*), qui est un mode d'adressage élaboré de ressources (utilisé notamment pour le Web). Plus précisément, l'URI est un élément générique qui se décline en trois sous-ensembles :

- l'URN (*Uniform Resource Name*) qui permet un nommage unique et permanent (même si la ressource devient inaccessible) ;
 - l'URC (*Uniform Resource Characteristic*) qui décrit les caractéristiques de la ressource ;
 - l'URL qui donne sa localisation.
-

/// XML-RPC

XML-RPC (*Remote Procedure Call*) est un protocole qui a été conçu pour permettre à des structures de données complexes d'être transmises, exécutées et renvoyées très facilement sur des plates-formes hétérogènes. XML-RPC est l'ancêtre de Soap.

► <http://www.xmlrpc.com/>

Pour éviter les commandes impayées liées à des numéros de cartes bancaires erronés, la société YAPS fait appel à la banque BarkBank pour valider ces numéros. Cette validation s'effectue lorsque les clients passent une commande en ligne. Les données sont échangées entre les deux sociétés au format XML et sont transportées par le protocole HTTP. Ces données contiennent les informations de la carte bancaire et sont envoyées à la société BarkBank, celle-ci effectue la validation et renvoie un statut : valide ou non (voir cas d'utilisation « Acheter des articles »).

Pour le transport des animaux, YAPS fait aussi appel à une société externe. Elle envoie des données à PetEx lui permettant d'acheminer d'un point vers un autre une commande d'animaux domestiques (voir le cas d'utilisation « Créer un bon de commande » du premier chapitre).

Ces échanges se font via des services web.

Les standards autour des services web

Les services web s'appuient sur un ensemble d'API et de protocoles standardisant l'invocation de composants applicatifs. Ils sont décrits en XML par des documents WSDL qui précisent les méthodes pouvant être invoquées, leurs signatures et les points d'accès du service (URL, port). UDDI est le standard de recherche de ces services et Soap le protocole utilisé pour échanger ces informations. Java EE 5 a considérablement simplifié le développement des services web. Nous n'aurons donc pas à manipuler directement tous ces standards. Il est cependant important d'en connaître la définition et leur rôle dans l'architecture globale des services web.

Soap

Inspiré de XML-RPC, Soap (*Simple Object Access Protocol*) est le protocole utilisé par les services web. Fondé sur XML, ce protocole autorise l'interopérabilité avec différents environnements logiciels quelle que soit leur plate-forme d'exécution (comme DCOM ou Corba IIOP). En effet, Soap permet la transmission de messages entre objets distants, en invoquant des méthodes sur des objets physiquement situés sur une autre machine. Le transfert se fait le plus souvent à l'aide du protocole HTTP.

Le protocole Soap se décompose en deux parties :

- une enveloppe, contenant des informations sur le message lui-même afin de permettre son acheminement et son traitement ;
- un modèle de données, définissant le format du message, c'est-à-dire les informations à transmettre.

Après avoir présenté le protocole Soap, décrivons plus précisément sa manipulation. En premier lieu, il est important de noter qu'en pratique Soap ne se manipule pas directement. En effet, en Java EE 5, l'API utilisée pour échanger des messages est JAX-WS (*Java API for XML-based Web Services*). Cela nous permet donc de nous concentrer sur le développement et l'invocation de services web, sans nous soucier du protocole Soap.

UDDI

UDDI (*Universal Description, Discovery and Integration*) est une spécification définissant la manière de publier et de retrouver des services web. C'est un annuaire qui offre des mécanismes d'enregistrement et de recherche de services web développés et publiés par des entreprises. UDDI fournit des informations sur l'auteur de services web (adresse, contact...), sur la classification (société informatique, hôpital...) et sur les moyens techniques permettant de les invoquer. Chaque entreprise peut avoir son propre annuaire UDDI pour publier ses services, mais elle peut aussi utiliser des annuaires publics comme ceux de Microsoft, IBM ou SAP.

UDDI n'est pas fondamentale aux services web comme l'est XML, Soap ou WSDL. Nous n'aborderons donc pas la publication de services dans cet ouvrage.

WSDL

WSDL (*Web Service Description Language*) est le format XML spécifié par le W3C permettant de définir un service web qui utilise le protocole Soap.

On expose ainsi au format XML la signature d'un service web accessible sur Internet. Cette signature inclut les opérations exposées, le type de ses paramètres d'entrées-sorties, et l'adresse réseau à laquelle on pourra l'invoquer. UDDI permet de retrouver un service web, et WSDL de décrire ses méthodes.

En fait, WSDL est scindé en deux parties qu'on appelle abstraite et concrète. La signature du service, ses méthodes et ses paramètres sont décrits de manière abstraite. Cette partie est ensuite liée à un protocole de communication et à un format de messages concrets. Ainsi, la partie abstraite est totalement découplée de la manière concrète permettant d'appeler le service.

APPROFONDIR Soap

- ▶ <http://www.w3.org/TR/soap/>
- 📖 Annick Fron, *Architectures réparties en Java : RMI, CORBA, JMS, sockets, SOAP, services web*, Dunod, 2007
- 📖 Jean-Marie Chauvet, *Services Web avec Soap, WSDL, UDDI, ebXML*, Eyrolles, 2002
- 📖 Hubert Kadima, Valérie Monfort, *Les Web services – Techniques, démarches et outils : XML, WSDL-, SOAP, UDDI, Rosetta, UML*, Dunod, 2003
- 📖 Robert Englander, *Java et SOAP*, O'Reilly, 2003

⚡ Pages blanches, jaunes et vertes

Un annuaire UDDI est constitué de pages blanches (nom de l'entreprise, adresse, contacts), jaunes (secteurs d'affaires relatifs au web service) et vertes (informations techniques des services web proposés).

APPROFONDIR UDDI

- ▶ <http://uddi.xml.org>
- ▶ <http://uddi.sap.com>
- ▶ <http://uddi.ibm.com/ubr/registry.html>
- ▶ <http://uddi.microsoft.com>

APARTÉ WS-I

WS-I, ou *Web Services Interoperability*, est un consortium d'industriels promouvant l'interopérabilité des services web au travers d'une implémentation nommée WS-I Basic Profile. Il édite aussi des guides de bonnes pratiques, des outils permettant de tester la conformité de services avec ses recommandations, ainsi que des forums de discussion dédiés aux développeurs.

- ▶ <http://www.ws-i.org/>

⚡ W3C

Le World Wide Web Consortium (W3C) est un consortium promouvant la compatibilité des technologies web telles que HTML, XHTML, XML, RDF, CSS, Soap, WSDL, UDDI, etc.

APPROFONDIR WSDL

- ▶ <http://www.w3.org/TR/wsdl>
- ▶ <http://www.w3schools.com/wsdl/default.asp>
- 📖 De Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey et Donald F. Ferguson, *Web Services Platform Architecture*, Prentice Hall, 2005

XML et XSD

Les services web sont intimement liés à XML et XSD (*XML Schema Definition*). Si vous n'êtes pas familier avec ce formalisme, consulter les références suivantes :

- 📖 Antoine Lonjon, Jean-Jacques Thomasson et Libero Maesano, *Modélisation XML*, Eyrolles, 2006
- 📖 Renaud Fleury, *Java/XML*, Eyrolles, 2004
- ▶ <http://www.w3.org/XML/>
- ▶ <http://www.w3.org/XML/Schema>

Extrait de fichier WSDL

```
<definitions targetNamespace="http://validator.barkbank.com/"
  ① name="ValidationService"> ②
  <types>
    <xsd:schema>
      <xsd:import namespace="http://validator.barkbank.com/"
        schemaLocation="http://localhost:8080/barkbank/
          ValidationService?xsd=1"/>
    </xsd:schema>
  </types>
  <message name="ValidateCard">
    <part name="parameters" element="tns:ValidateCard"/>
  </message>
  (...)
  <service name="ValidationService"> ③
    <port name="ValidationPort"
      binding="tns:ValidationPortBinding">
      <soap:address location=
        "http://localhost:8080/barkbank/ValidationService"/> ④
    </port>
  </service>
</definitions>
```

Cet extrait de document WSDL commence par l'en-tête **definitions** ①. Cet élément peut prendre plusieurs attributs facultatifs qui définissent des noms de domaines à utiliser dans la suite du document. Dans notre exemple, la définition reçoit le nom **ValidationService** ②. Le service web portant ce même nom ③ peut être invoqué à partir d'une URL donnée ④.

Ce document utilise certains noms de domaines définis à l'aide de préfixes :

- **tns** : abrégé de « this namespace » (ce nom de domaine), c'est-à-dire le nom de domaine d'accueil contenant le service ;
- **xsd** : nom de domaine du schéma XML (XSD) utilisé pour définir les types du document ;
- **soap** : nom de domaine utilisé pour la liaison Soap.

Nous ne nous attarderons pas plus sur WSDL car, comme vous le découvrirez plus loin, ce document est généré automatiquement et n'a pas à être développé manuellement.

JAX-WS 2.0

JAX-WS est la nouvelle appellation de JAX-RPC (*Java API for XML Based RPC*) qui permet de développer très simplement des services web. JAX-WS fournit un ensemble d'annotations pour mapper la correspon-

RAPPEL Sigles et acronymes

Vous trouverez en annexe un lexique d'acronymes et de sigles.

dance Java-WSDL. Il suffit pour cela d'annoter directement les classes Java. En ce qui concerne le client, JAX-WS permet d'utiliser une classe proxy pour appeler un service distant et masquer la complexité du protocole. Ainsi, ni le client ni le serveur n'ont besoin de générer ou de parser les messages Soap. JAX-WS s'occupe de ces traitements bas niveau.

Exemple d'annotations JAX-WS dans une classe Java

```
@WebService
public class Validation {

    @WebMethod
    public String validateCreditCard() {
        (...)
    }
}
```

Dans l'exemple ci-dessus, une classe Java utilise des annotations JAX-WS qui vont permettre par la suite de générer un document WSDL.

JAXB 2.0

JAX-WS s'appuie sur l'API JAXB 2.0 pour tout ce qui concerne la correspondance entre documents XML et objets Java. JAXB (*Java Architecture for XML Binding*) facilite cette correspondance bidirectionnelle en fournissant un niveau d'abstraction plus élevé que SAX ou DOM et en s'appuyant sur des annotations.

L'API JAXB, définie dans le paquetage `javax.xml.bind`, fournit un ensemble d'interfaces et de classes que les applications utilisent pour produire un schéma XML, générer des classes Java ou des documents XML. Les types de manipulations sont les suivantes (figure 9-1).

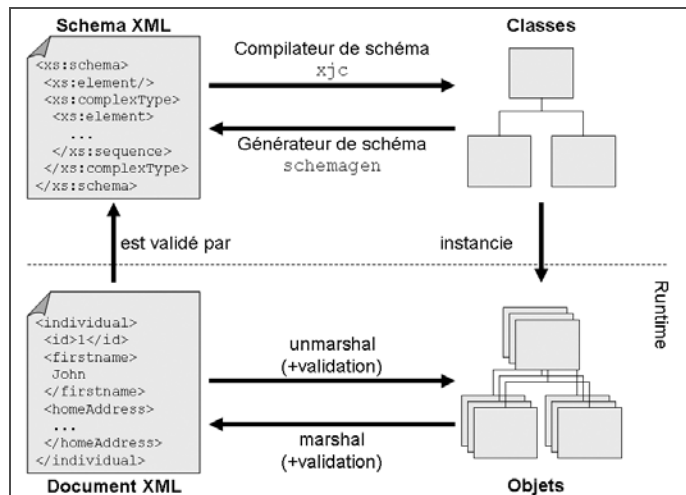


Figure 9-1
Les phases de manipulation JAXB

ARCHITECTURE Design pattern Proxy

Le Proxy, très utilisé pour la gestion d'objets distribués, ajoute un niveau de redirection vers une méthode d'un objet. L'idée est de construire un Proxy capable de communiquer avec un objet distant sans que l'appelant fasse de différences entre un accès local ou un accès distant.

/// JAX-RPC

JAX-RPC (*Java API for XML-based Remote Procedure Call*) est une API permettant de créer des services et clients web basés XML et RPC.

- <http://java.sun.com/webservices/jaxrpc/index.jsp>

APPROFONDIR JAX-WS

- <http://java.sun.com/webservices/jaxws/index.jsp>
- <http://jax-ws.dev.java.net/>

APPROFONDIR JAXB

- <https://jaxb.dev.java.net/>
- <http://java.sun.com/webservices/docs/2.0/tutorial/doc/JAXBWorks.html>

Pour savoir comment transformer un graphe d'objets en XML en utilisant JAXB, vous pouvez consulter un article que j'ai écrit pour le site DevX :

- <http://www.devx.com/Java/Article/34069>

- Grâce à son compilateur de schéma, JAXB peut générer un ensemble de classes Java à partir d'un schéma XML.
- L'action inverse est possible grâce à un générateur de schéma. À partir d'une grappe de classes Java, JAXB peut créer un schéma XML.
- La classe `Unmarshaller` transforme un document XML en un graphe d'objets.
- La classe `Marshaller` fabrique un graphe d'objets à partir d'un document XML.

Toutes ces fonctionnalités font que JAXB est souvent associé aux services web. Les services web utilisent l'API JAXB pour transformer des objets en messages XML et inversement.

Par exemple, si on veut obtenir une représentation XML de la classe `Customer`, il suffit de l'annoter avec `@javax.xml.bind.annotation.XmlRootElement` ❶. D'autres annotations permettent de spécifier qu'un attribut est un identifiant ❷ ou de renommer ❸ un attribut (e-mail au lieu de email, par exemple).

Exemple d'annotations JAXB dans la classe `Customer`

```
@XmlRootElement ❶
public class Customer {

    @XmlID ❷
    private String id;
    private String firstname;
    private String lastname;
    @XmlAttribute(name = "e-mail") ❸
    private String email;
    (...)
}
```

Ces annotations permettent alors de générer le document XML suivant à partir de la classe, et inversement.

Représentation XML de la classe `Customer`

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<customer> ❶
  <id>1234</id> ❷
  <firstname>Paul</firstname>
  <lastname>Smith</lastname>
  <e-mail>yaps@petstore.com</e-mail> ❸
</customer>
```

SAX et DOM

Il existe deux grandes familles de solutions pour lire un fichier XML en Java :

- **SAX** (*Simple API for XML*) est une solution très pratique pour parser des fichiers XML simples. L'analyseur SAX s'occupe d'interpréter le document XML, identifiant les différents attributs et balises. Dès qu'un élément est reconnu, SAX avertit le programme afin que celui-ci fasse un traitement approprié. Le programme reçoit donc des événements du type « ouverture de la balise X », « zone de texte », « fermeture de la balise X ». Après le passage de SAX, seules restent les informations que le programmeur a lui-même traitées et placées dans d'autres structures de données.
- **DOM** (*Document Object Model*) est plus simple à comprendre puisqu'il lit l'intégralité du document XML afin de construire sa représentation en mémoire. Ensuite, le programme peut librement et rapidement parcourir cette représentation interne pour y garder les informations souhaitées. DOM n'est pas recommandé pour des documents de taille importante à cause de l'occupation mémoire que les données pourraient prendre.

Services web

Souvent décrits comme le nouveau modèle de référence pour les systèmes d'information, les services web permettent à des applications de dialoguer à distance, et ceci indépendamment des plates-formes et des langages sur lesquels elles reposent. Pour communiquer, les services web s'appuient sur les standards que nous venons de voir. Cette communication est basée sur le principe de demandes et de réponses (en fait, des messages XML) transportés par le protocole HTTP.

Les services web sont aujourd'hui incontournables et se présentent comme le nouveau paradigme des architectures logicielles ou architectures orientées services (SOA). Cette technologie tend à s'imposer comme le nouveau standard en termes d'intégration et d'échanges B2B. Grâce aux services web, les applications peuvent être vues comme un ensemble de services métier, structurés et dialoguant selon un standard, plutôt qu'un ensemble d'objets et de méthodes.

SOA

Le terme SOA (*Service-Oriented Architecture*) définit une architecture logicielle à base de services. Un service désigne une action exécutée par un composant fournisseur à l'attention d'un composant consommateur.

Exemple de service web

La plate-forme Java EE 5 cache tous les mécanismes de bas niveau et rend l'écriture d'un service web extrêmement simple. En effet, il suffit d'une seule annotation JAX-WS ❶ pour transformer une simple classe Java en service web.

Exemple simple de service web

```
@WebService ❶
public class Validation {

    public String validateCreditCard(String ccNumber,
                                     String ccType,String ccExpiryDate) {

        (...)
        return "OK";
    }
}
```

Exemple d'appel à un service web

Il est tout aussi facile de consommer un service web. Bien que la tuyauterie technique utilise des classes générées, des proxy pour communiquer à distance et toute la panoplie de spécifications que nous venons de voir (WSDL, Soap et JAXB), une simple annotation **@WebServiceRef** ❷ permet d'injecter une référence à un service web.

Exemple simple d'appel à un service web

```
public class ShoppingCartController {

    @WebServiceRef ❷
    private ValidationService validationService;

    private void validateCreditCard(CreditCard creditCard) {
        Validation validation=validationService.getValidationPort();
        validation.validateCard(creditCard.getCreditCardNumber(),
                               creditCard.getCreditCardType(),
                               creditCard.getCreditCardExpDate());
    }
}
```

Annotations JAX-WS

Les annotations JAX-WS sont spécifiques aux services web. Elles permettent d'agir sur la structure du document WSDL en modifiant certains paramètres du service ou des méthodes qui le composent. Dans cette section, nous allons décrire plus finement le comportement de ces annotations.

Le service

L'annotation principale pour définir un service web est `@javax.jws.WebService`. Elle utilise plusieurs attributs et peut, par exemple, spécifier la localisation d'un service web.

Code de l'annotation `@javax.jws.WebService`

```
package javax.jws;

@Target({TYPE}) @Retention(RUNTIME)
public @interface WebService {

    String name() default "";

    String targetNamespace() default "";
    String serviceName() default "";
    String wsdlLocation() default "";
    String portName() default "";
    String endpointInterface() default "";
}
```

- ◀ Cette annotation s'applique à une classe.
- ◀ Nom donné au service web mappé sur l'élément `portType` du WSDL.
- ◀ Spécifie le namespace XML utilisé dans le WSDL.
- ◀ URL d'accès au WSDL définissant le service.

La méthode

Si une classe est annotée par `@WebService`, alors par défaut, toutes ses méthodes publiques peuvent être appelées. Si on veut restreindre cette règle et ne publier que certaines méthodes, on peut alors utiliser l'annotation `@javax.jws.WebMethod`. Celle-ci permet aussi de modifier certains attributs par défaut.

Code de l'annotation `@javax.jws.WebMethod`

```
package javax.jws;

@Target({METHOD}) @Retention(RUNTIME)
public @interface WebMethod {

    String operationName() default "";

    boolean exclude() default false;

    String action() default "";
}
```

- ◀ Cette annotation s'applique à une méthode.
- ◀ Redéfinit le nom de la méthode.
- ◀ Ne publie pas la méthode si `exclude=true`.
- ◀ Valeur de l'action Soap.

Une méthode qui n'a pas de paramètres de retour peut être annotée par `@OneWay`.

Code de l'annotation `@javax.jws.OneWay`

```
package javax.jws;

@Target({METHOD}) @Retention(RUNTIME)
public @interface OneWay {}
```

- ◀ Cette annotation s'applique à une méthode.

L'exemple suivant redéfinit le nom de la méthode en `Deliver` au lieu de `deliverItems` ❶ et, comme elle ne possède pas de paramètres de retour, elle est annotée `@OneWay` ❷.

Service web redéfinissant le nom de la méthode

```
@WebService
public class Delivery {

    @WebMethod(operationName = "Deliver") ❶
    @Oneway ❷
    public void deliverItems(DeliveryPlace from,
                           DeliveryPlace to, String reference) {
        (...)
    }
}
```

Les paramètres de la méthode

Les paramètres de la méthode ainsi que la valeur de retour peuvent aussi être changés. L'annotation `@javax.jws.WebParam` permet de contrôler la génération du WSDL qui concerne les paramètres de la méthode.

Code de l'annotation `@javax.jws.WebParam`

Cette annotation s'applique à un paramètre.
Redéfinit le nom du paramètre.
Nom du wsdl : part qui représente le paramètre.
Namespace XML du paramètre.
Indique que le paramètre doit être mis dans l'en-tête (header) de Soap et non dans le corps (body).
Indique si le paramètre est en entrée (IN), sortie (OUT), ou les deux (INOUT).

```
package javax.jws;

@Target({PARAMETER}) @Retention(RUNTIME)
public @interface WebParam {

    String name() default "";

    String partName() default "";

    String targetNamespace() default "";

    boolean header() default false;

    Mode mode() default IN;
}
```

L'annotation `@javax.jws.WebResult` est presque identique, mais elle annote la valeur de retour de la méthode.

Code de l'annotation `@javax.jws.WebResult`

Cette annotation s'applique à une méthode.
Définit le nom de la valeur de retour.

```
package javax.jws;

@Target({METHOD}) @Retention(RUNTIME)
public @interface WebResult {

    String name() default "";
}
```

```
String partName() default "";

String targetNamespace() default "";

boolean header() default false;
}
```

- ◀ Nom du `wsdl:part` qui représente la valeur de retour.
- ◀ Namespace XML de la valeur de retour.
- ◀ Indique que la valeur de retour doit être mise dans l'en-tête (header) de Soap et non dans le corps (body).

Le code suivant utilise ces deux annotations pour changer le nom des paramètres (`expiryDate` au lieu de `ccExpiryDate` ❸) ou leur mode ❷ (`WebParam.Mode.IN`). L'annotation `@WebResult(name = "cardStatus")` ❸ permet de nommer la valeur de retour qui est de type `String`.

Service web redéfinissant les paramètres de la méthode

```
@WebService
public class Validation {

    @WebMethod
    @WebResult(name = "cardStatus") ❶
    public String validateCreditCard(
        @WebParam(name = "creditCardNumber",
            ❷ mode = WebParam.Mode.IN)String ccNumber,
        @WebParam(name = "creditCardType")String ccType,
        @WebParam(name = "expiryDate")String ccExpiryDate) ❸{
        (...)
    }
}
```

Pour mieux comprendre l'utilité de ces annotations, regardez attentivement les deux versions de schéma XML. Ces schémas représentent les paramètres de la méthode `validateCreditCard` avec et sans annotations JAX-WS.

REMARQUE Les fichiers XSD

Comme nous le verrons par la suite, les fichiers XSD sont générés automatiquement puis déployés avec les classes du service web.

Extrait du schéma XSD des paramètres de la méthode sans annotations

```
<xs:complexType name="validateCreditCard"> ❶
  <xs:sequence>
    <xs:element name="arg0" type="xs:string" minOccurs="0"/> ❷
    <xs:element name="arg1" type="xs:string" minOccurs="0"/> ❷
    <xs:element name="arg2" type="xs:string" minOccurs="0"/> ❷
  </xs:sequence>
</xs:complexType>

<xs:complexType name="validateCreditCardResponse">
  <xs:sequence> ❸
    <xs:element name="return" type="xs:string" minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```


/// Artefact

Un artefact est composé de l'ensemble des documents nécessaires à un service web. On peut citer par exemple le document WSDL, ou encore les classes Java qui formeront les messages d'échanges XML.

PRÉCISION Service web et EJB

Il y a deux moyens d'implémenter un service web. Le premier repose sur les servlets où une simple classe annotée est déployée dans un conteneur web (dans un WAR). L'autre moyen repose sur les EJB sans état qui sont annotés à la fois par `@Stateless` et `@WebService`, puis déployés dans un conteneur EJB (dans un JAR ou EAR). Dans cet ouvrage, nous utiliserons la première solution.

Extrait du schéma XSD des paramètres de la méthode avec annotations

```
<xs:complexType name="ValidateCard"> ❶
<xs:sequence>
  <xs:element name="creditCardNumber" type="xs:string" minOccurs="0"/> ❷
  <xs:element name="creditCardType" type="xs:string" minOccurs="0"/> ❷
  <xs:element name="expiryDate" type="xs:string" minOccurs="0"/> ❷
</xs:sequence>
</xs:complexType>

<xs:complexType name="ValidateCardResponse">
<xs:sequence> ❸
  <xs:element name="cardStatus" type="xs:string" minOccurs="0"/>
</xs:sequence>
</xs:complexType>
```

Le nom de la méthode ❶ peut être modifié en utilisant l'annotation `@WebMethod`. Idem pour le nom des paramètres ❷ (avec `@WebParam`) et du type de retour ❸ (`@WebResult`).

Comment développer un service web

Comme nous venons de le voir dans ces quelques exemples, le développement d'un service web est relativement simple bien que plusieurs technologies soient mises en œuvre. Telle est la force de Java EE 5. Pour cela, il y a plusieurs phases de génération de code qui entrent en jeu et qui mettent en œuvre toute la tuyauterie technique. Le développement et l'utilisation d'un service web comportent quatre phases :

- développement du service web ;
- génération des artefacts serveurs ;
- génération des artefacts clients ;
- appel du service web.

Développer la classe du service web

Prenons l'exemple du service web de la BarkBank qui permet de valider une carte bancaire. La classe `Validation` ❶ publie une méthode ❷ `validateCreditCard` (renommée en `ValidateCard` ❷) qui prend en paramètres le numéro de la carte, son type (Visa, Master Card, etc.) et la date d'expiration. Tous ces paramètres ❸ sont de type `String` ainsi que la valeur de retour ❹ correspondant au statut (numéro invalide, carte expirée, etc.).

Voilà, tout est dit. N'est-ce pas magique ? Si les valeurs par défaut vous conviennent, vous pouvez même limiter les annotations à la seule `@WebService`. Contrairement aux EJB, un service web n'a pas besoin d'implémenter une interface.

```

@WebService
public class Validation { ❶

    @WebMethod(operationName = "ValidateCard") ❷
    @WebResult(name = "cardStatus") ❸
    public String validateCreditCard( ❹
        @WebParam(name = "creditCardNumber")String ccNumber,
        @WebParam(name = "creditCardType")String ccType, ❺
        @WebParam(name = "expiryDate")String ccExpiryDate) {
        // L'algorithme de vérification n'est pas détaillé
    }
}

```

Générer les artefacts serveurs

Cette classe développée, BarkBank doit générer les artefacts de son service, c'est-à-dire le document WSDL et les classes Java qui formeront les messages d'échanges XML. Pour cela, on utilise l'utilitaire `wsgen` fourni avec GlassFish. À partir de la classe `Validation`, l'utilitaire `wsgen` génère les éléments suivants :

- La classe `ValidateCreditCard` (du nom de la méthode du service) correspond aux paramètres passés au service. Cette classe ne contient que les trois attributs de type `String` (numéro, type et date d'expiration) ainsi que les accesseurs. Comme elle est responsable du transport des paramètres au format XML, cette classe utilise les annotations JAXB.
- La classe `ValidateCreditCardResponse` (du nom de la méthode suffixée par `Response`) correspond à la valeur de retour. Cette classe est, elle aussi, annotée par JAXB.
- Le document WSDL décrivant le service web et son schéma XSD.

Ci-après le diagramme de classes représentant le service web `Validation` ainsi que les artefacts générés dans le sous-paquetage `jaxws` (figure 9-2).

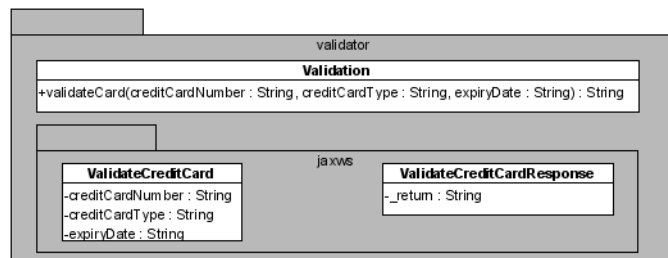


Figure 9-2
Artefacts serveur du service web de validation

Bien que vous n'ayez pas à vous soucier des classes générées, il est intéressant de voir un extrait de leur contenu.

ANNOTATION **@OneWay**

Les méthodes annotées `@OneWay` n'ont pas de valeur de retour. Il n'y a donc pas de classes `Response` générées pour ce type de méthodes.

La classe `ValidateCreditCard` représente les paramètres de la méthode de validation. Elle utilise les annotations `JAXB @XmlElement` et `@XmlElement` pour générer un message XML. Les noms des attributs (ex. `expiryDate`) correspondent aux noms spécifiés dans les annotations JAX-WS du service web :
`@WebParam(name = "expiryDate")`
`String ccExpiryDate`

La classe `ValidateCreditCardResponse` représente la valeur de retour. Le nom `cardStatus` fait référence à l'annotation JAX-WS du service web :
`@WebResult(name = "cardStatus")`

Classe générée correspondant aux paramètres

```

@XmlRootElement(name = "ValidateCard")
public class ValidateCreditCard {

    @XmlElement(name = "creditCardNumber")
    private String creditCardNumber;
    @XmlElement(name = "creditCardType")
    private String creditCardType;
    @XmlElement(name = "expiryDate")
    private String expiryDate;
}

```

Classe générée correspondant aux valeurs de retour

```

@XmlRootElement
public class ValidateCreditCardResponse {

    @XmlElement(name = "cardStatus")
    private String _return;
}

```

TÉLÉCHARGER Le code généré

Retrouvez la totalité des sources générées sur le site YAPS Pet Store à l'adresse suivante :
 ▶ <http://www.antoniogoncalves.org>

Générer les artefacts clients

BarkBank a développé son service web, généré ses artefacts serveurs et a déployé le tout sur son serveur à une URL donnée. Maintenant, pour que YAPS puisse accéder à ce service, il lui faut générer les artefacts côté client. Cette opération se fait via l'utilitaire `wsimport`. Celui-ci prend en paramètres l'URL du WSDL du service web. À partir de ce WSDL, `wsimport` génère les éléments suivants :

- `ValidationService` est la classe principale qui est utilisée dans le code de l'application YAPS Pet Store. Celle-ci retourne l'interface `Validation`, qui possède la même signature que le service web.
- Une fabrique (`ObjectFactory`) pour créer les deux mêmes classes `ValidateCard` et `ValidateCardResponse`. Grâce aux annotations JAXB, ces classes génèrent les messages XML.

Le diagramme ci-après nous montre les différentes classes et interfaces générées et utilisées par le client pour invoquer le service web (figure 9-3).

Appeler un service web

Une fois le service web déployé et les artefacts clients générés, il est temps de les utiliser pour invoquer le service web depuis l'application YAPS Pet Store. Cette tâche, assez complexe en J2EE, s'est considérablement simplifiée en Java EE 5 grâce à l'injection et à l'annotation `@javax.xml.ws.WebServiceRef`. En effet, il suffit d'annoter une classe et d'utiliser les artefacts générés pour appeler un service web.

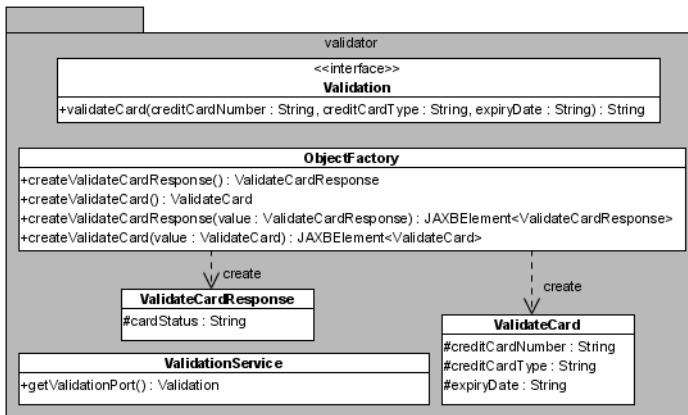


Figure 9–3
Artefacts client du serveur web de validation

Extrait du code appelant le service web

```

public class ShoppingCartController {

    @WebServiceRef
    private ValidationService validationService;

    private void validateCreditCard(CreditCard creditCard) {
        Validation validation=validationService.getValidationPort();

        String statusCard = validation.validateCard(
            creditCard.getCreditCardNumber(),
            creditCard.getCreditCardType(),
            creditCard.getCreditCardExpDate());

        (...)
    }
}

```

- ◀ Le service web peut-être appelé à partir de n'importe quelle classe (un managed bean dans notre cas).
- ◀ La classe ValidationService est générée par wsimport. L'annotation @WebServiceRef permet d'injecter la référence du service web.
- ◀ On récupère l'interface Validation qui a été générée par wsimport.
- ◀ Appel du service web en lui passant les paramètres et en récupérant la valeur de retour.

L'annotation @WebServiceRef peut prendre plusieurs paramètres. Dans l'exemple précédent, nous aurions pu lui fournir l'URL du document WSDL de la manière suivante

```

@WebServiceRef(wsdlLocation =
    "http://localhost:8080/barkbank/ValidationService?WSDL")
private ValidationService validationService;

```

Code de l'annotation @javax.xml.ws.WebServiceRef

```

package javax.xml.ws;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface WebServiceRef {

    String name() default ""
}

```

- ◀ Cette annotation s'applique à une méthode ou à un attribut.
- ◀ Nom local du service web.

URL du document WSDL décrivant le service web.

Nom local du service web spécifique au serveur d'applications.

Au lieu de spécifier l'URL du service via `wsdlLocation`, on peut directement spécifier le nom de l'interface générée.

Code de l'annotation `@javax.xml.ws.WebServiceRef`

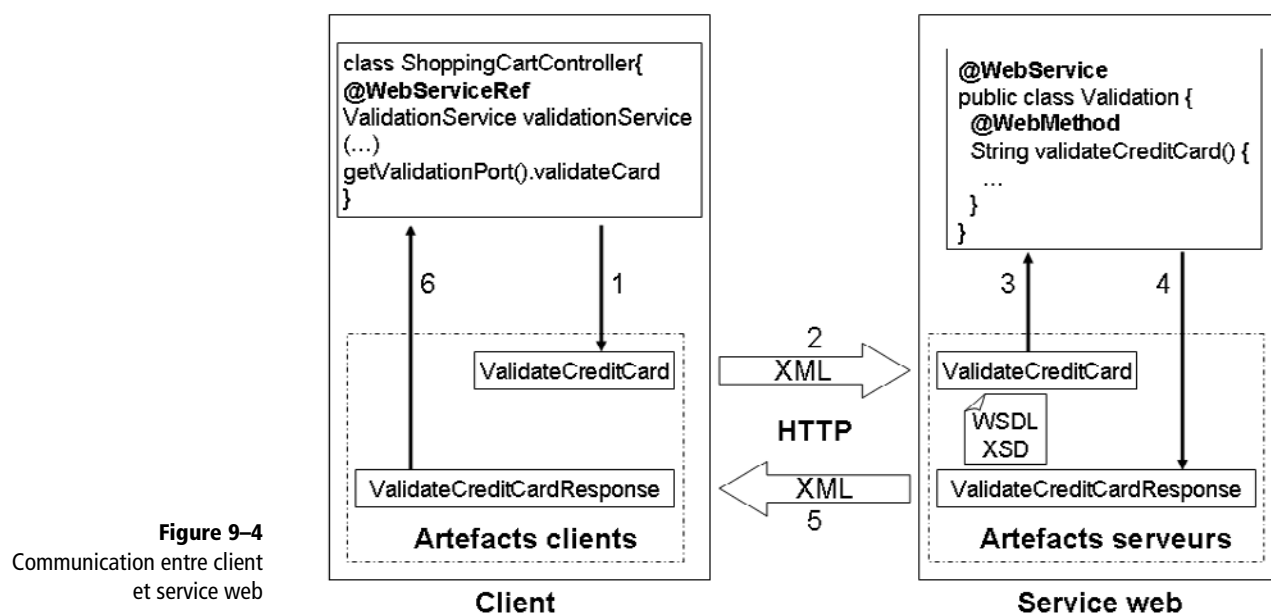
```
String wsdlLocation() default "";

String mappedName() default "";

Class type() default Object.class;
Class value() default Object.class;
}
```

La vision globale

Rien ne vaut un schéma pour éclaircir le mécanisme d'invocation d'un service web (figure 9-4).



Pour que le client puisse invoquer le service web, on annote par `@WebServiceRef` la classe générée `ValidationService` (`@WebServiceRef` permet au conteneur d'injecter une référence du service). On appelle ce service en lui passant le numéro de carte de crédit, le type et la date d'expiration. Ceci a pour effet d'affecter ces données dans la classe `ValidateCreditCard` ①. Grâce aux annotations JAXB, cette classe génère un document XML qui peut alors transiter par HTTP ② dans une enveloppe Soap. À la réception de ce message XML, le service web utilise les annotations JAXB pour reconstruire un objet `ValidateCreditCard` ③. Il valide alors les données bancaires puis retourne le résultat de cette validation via la classe

`ValidateCreditCardResponse` ④. En utilisant le même mécanisme, cette classe est transformée en flux XML pour transiter à travers le réseau ⑤. Le résultat arrive enfin chez le client qui le retransforme en objet ⑥ et peut ainsi récupérer la valeur de retour.

Les services web utilisés par YAPS Pet Store

L'application YAPS Pet Store utilise deux services web : la validation des numéros de cartes bancaires de la société BarkBank et le service de transport de marchandises de PetEx.

La validation des cartes de crédit

Pour vérifier que les coordonnées bancaires sont exactes, YAPS a passé un partenariat avec la banque BarkBank. Celle-ci possède un service web de validation de cartes bancaires. Ce service prend en paramètres le numéro de la carte, son type (Visa, MasterCard, etc.) et sa date d'expiration au format MM/AA (mois/année). L'algorithme de vérification s'assure que la date d'expiration est supérieure à la date du jour, et que le numéro des cartes Visa est pair.

REMARQUE L'algorithme de vérification

Comme pour l'existence de la société YAPS, l'algorithme de vérification des cartes bancaires est complètement fictif.

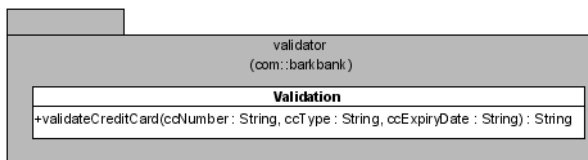


Figure 9-5
Diagramme de classe
du service web de validation

Code du service web Validation

```

@WebService
public class Validation {

    @WebMethod(operationName = "ValidateCard")
    @WebResult(name = "cardStatus")
    public String validateCreditCard(
        @WebParam(name = "creditCardNumber")String ccNumber,
        @WebParam(name = "creditCardType")String ccType,
        @WebParam(name = "expiryDate")String ccExpiryDate) {

        Calendar calendar = Calendar.getInstance();
        int year = getExpiryYear(ccExpiryDate);
        int month = getExpiryMonth(ccExpiryDate);
        int lastNumber = getLastNumber(ccNumber);
    }
}
  
```

◀ La méthode de validation prend en paramètre le numéro de la carte, son type, sa date d'expiration, et retourne le statut de la carte.

◀ On utilise des méthodes utilitaires pour obtenir l'année et le mois à partir du format MM/AA.

- L'année de la carte est inférieure à l'année en cours. On renvoie le statut « L'année de la carte bancaire est expirée ».

L'année de la carte est bonne, mais le mois est inférieur au mois en cours. On renvoie le statut « Le mois de la carte bancaire est expiré ».

La carte est valide.

Code du service web Validation

```
if (year < calendar.get(Calendar.YEAR)) {
    return "The year of the credit card is passed";
}

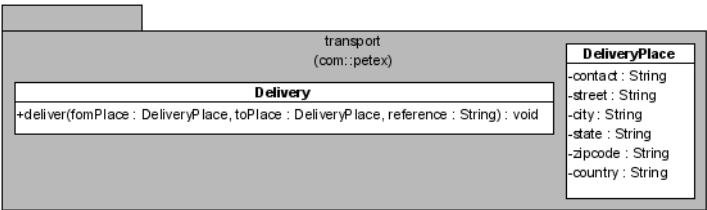
if (year == calendar.get(Calendar.YEAR)
    && month < calendar.get(Calendar.MONTH)) {
    return "The month of the credit card is passed";
}

return "OK";
}
```

Avertir le transporteur

Pour la livraison des marchandises, YAPS utilise le transporteur PetEx. Celui-ci possède un service web lui permettant d'être averti des transports à effectuer. Ce service prend trois paramètres : une adresse de départ, lieu où la marchandise est chargée dans les camions, une adresse de livraison et une référence. Dans notre cas, cette référence correspond au numéro de la commande.

Figure 9-6
Diagramme de classe
du service web de livraison



Notez dans le diagramme ci-dessus que la méthode `deliver` prend un objet en paramètre (`DeliveryPlace`). Cet objet correspond à une adresse qui permet à PetEx de charger ou de livrer une marchandise. On voit ici que les services web ne sont pas obligés d'utiliser seulement des types primitifs. De toute façon, lors de l'utilisation de `wsgen` et `wsimport`, des artefacts vont être générés pour annoter la classe `DeliveryPlace` avec JAXB. Cet objet sera donc transformé en flux XML.

Code du service web Delivery

- Ce service ne retourne pas de résultat. On peut donc utiliser l'annotation `@OneWay`.

```
@WebService
public class Delivery {
    private Logger logger=Logger.getLogger("com.petex.transport");

    @WebMethod
    @OneWay
    public void deliverItems(DeliveryPlace from,
                            DeliveryPlace to, String reference){
```

```

logger.info("Delivery Order Received");
logger.info("Deliver from " + from);
logger.info("Deliver to " + to);
logger.info("Reference n° " + reference);

```

```

}
}

```

Nous ne nous intéressons pas ici à la manière dont le système de PetEx est averti. Nous nous contentons juste d'afficher les informations par le biais d'un logger.

Appel des services web

Ces deux services web sont invoqués à différents endroits dans l'application YAPS Pet Store. La création d'un bon de commande fait intervenir plusieurs composants. Le client saisit son adresse de livraison et ses coordonnées bancaires à partir de la page `confirmorder.jsp`, puis il clique sur *Submit* (figure 9–6).

Figure 9–7 `confirmorder.jsp`

Le clic sur *Submit* invoque ❶ la méthode `confirmOrder()` du managed bean `ShoppingCartController` qui commence par invoquer le service web de validation des cartes ❷. Si les données sont valides, alors il appelle la

UML Stéréotypes BCE

Trois autres stéréotypes ont été intégrés à UML et sont souvent utilisés pour le pattern MVC :

`<<boundary>>` : (la vue) représente les objets qui réalisent les échanges entre le système et les acteurs comme les pages web ou les interfaces graphiques.

`<<control>>` : (le contrôleur) objet implémentant des mécanismes de collaboration comme les managed beans.

`<<entity>>` : (le modèle) objet représentant les informations du système comme les entités.

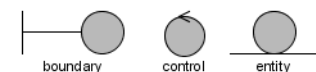
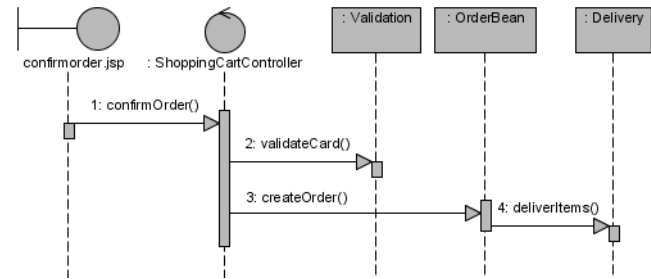


Figure 9–8
Diagramme de séquences
pour la création d'un bon de commande



Appel du service web Validation à partir du managed bean

- On annote par `@WebServiceRef` l'interface générée par `wsimport`. Le système d'injection se charge d'instancier l'objet.
- Cette méthode est appelée lorsque le client saisit ses données bancaires et soumet le formulaire.
- Appel d'une méthode privée pour invoquer le service web.
- S'il n'y a pas d'exceptions, on invoque le stateless bean pour créer le bon de commande.
- En cas d'exceptions, on affiche un message sur la page web.
- Méthode privée appelant le service web.
- Appel du service web en lui passant les paramètres.
- Si les données bancaires sont invalides (statut différent de OK), alors on lance une exception avec un message qui sera affiché à l'écran.

```
public class ShoppingCartController
{
    @WebServiceRef
    private ValidationService validationService;

    public String confirmOrder() {
        try {
            validateCreditCard(creditCard);

            order = orderBean.createOrder(customer, deliveryAddress,
                creditCard, shoppingCartBean.getCartItems());
            shoppingCartBean.empty()
        } catch (Exception e) {
            addMessage(e);
        }
        return "order.confirmed";
    }

    private void validateCreditCard(CreditCard creditCard) {
        Validation validation=validationService.getValidationPort();
        String statusCard = validation.validateCard(
            creditCard.getCreditCardNumber(),
            creditCard.getCreditCardType(),
            creditCard.getCreditCardExpDate());

        if (!"OK".equals(statusCard))
            throw new CreditCardException(
                "Credit Card is invalid : " + statusCard);
    }
}
```

Si la carte bancaire est valide, l'application appelle l'EJB stateless `OrderBean` pour créer un bon de commande. Une fois la création effectuée, l'EJB invoque le service web du transporteur pour l'avertir des livraisons qu'il doit faire.

Appel du service web `Delivery` à partir du stateless session bean

@Stateless

```
public class OrderBean implements OrderRemote, OrderLocal {
```

```
    @WebServiceRef
```

```
    private DeliveryService deliveryService;
```

```
    public Order createOrder(Customer customer,
                             Address deliveryAddress, CreditCard creditCard,
                             List<CartItem> cartItems) {
```

```
        Order order = new Order(customer,
                                   em.merge(deliveryAddress), creditCard);
```

```
        List<OrderLine> orderLines = new ArrayList<OrderLine>();
        for (CartItem cartItem : cartItems) {
            orderLines.add(new OrderLine(cartItem.getQuantity(),
                                          cartItem.getItem()));
        }
```

```
        order.setOrderLines(orderLines);
```

```
        em.persist(order);
```

```
        notifyTransporter(order);
```

```
        return order;
```

```
    }
```

```
    private void notifyTransporter(Order order) {
```

```
        DeliveryPlace from = new DeliveryPlace();
        from.setContact(Constants.COMPANY_NAME);
        from.setStreet(Constants.COMPANY_STREET);
        from.setCity(Constants.COMPANY_CITY);
        from.setState(Constants.COMPANY_STATE);
        from.setZipcode(Constants.COMPANY_ZIPCODE);
        from.setCountry(Constants.COMPANY_COUNTRY);
```

```
        DeliveryPlace to = new DeliveryPlace();
        to.setContact(order.getCustomer().getLastName());
        to.setStreet(order.getDeliveryAddress().getStreet1());
        to.setCity(order.getDeliveryAddress().getCity());
        to.setState(order.getDeliveryAddress().getState());
        to.setZipcode(order.getDeliveryAddress().getZipcode());
        to.setCountry(order.getDeliveryAddress().getCountry());
```

```
        Delivery delivery = deliveryService.getDeliveryPort();
        delivery.deliverItems(from, to, order.getId().toString());
```

```
    }
```

```
}
```

◀ Référence au service web.

◀ Pour créer un bon de commande, il est nécessaire de disposer du contenu du panier électronique (`cartItems`) mais aussi de l'adresse de livraison, de la carte bancaire et des références du client.

◀ Création d'un objet bon de commande.

◀ À partir du panier électronique, on crée des lignes de commande.

◀ L'objet bon de commande est rendu persistant.

◀ Appel d'une méthode privée pour invoquer le service web.

◀ Méthode privée appelant le service web.

◀ Le service du transporteur a besoin de connaître l'adresse de chargement et de livraison des articles.

◀ On invoque le service web en lui passant les deux adresses ainsi que le numéro de bon de commande.

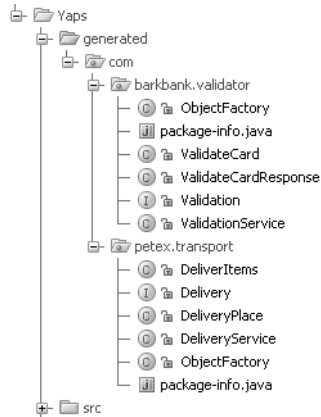


Figure 9-9 Classes générées pour l'application YAPS Pet Store

Paquetages des différents services web

Les services web étant hébergés par les partenaires BarkBank et PetEx, ils ne se trouvent pas dans l'application YAPS Pet Store. On utilise donc des projets différents pour accueillir ces sources. Pour distinguer les classes que l'on développe (src) de celles qui sont générées (generated) par wsgen ou wsimport, on se sert de répertoires différents. Ainsi, pour BarkBank, la classe du service web se trouve dans `src/com.barkbank.validator` alors que les artefacts sont générés dans `generated/com.barkbank.validator.jaxws`. Il en est de même pour PetEx.

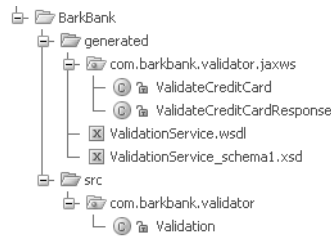


Figure 9-10 Service web de BarkBank

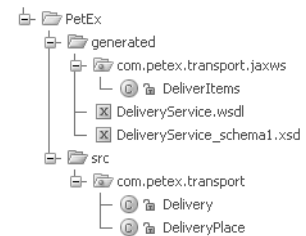


Figure 9-11 Service web de PetEx

L'application YAPS Pet Store utilise l'outil `wsimport` pour générer les artefacts côté client des deux services web. Ces classes se trouvent dans le répertoire `generated`.

Architecture

Le diagramme suivant nous montre comment les services web externes de BarkBank et PetEx s'insèrent dans l'architecture. Chaque service est invoqué par un type de composant différent de notre application. C'est un managed bean (`ShoppingCartController`) qui invoque le service de vérification de cartes, alors que c'est un stateless bean (`OrderBean`) qui se charge d'avertir le transporteur (figure 9-12).

Exécuter l'application

Pour simuler la réalité des applications distribuées, on aurait pu créer une instance GlassFish différente pour y déployer les applications de BarkBank, PetEx et YAPS Pet Store. Il aurait même été possible de déployer chaque application sur un serveur physique distinct et les faire communiquer au travers d'un réseau. Pour ne pas compliquer le déploiement et l'exécution de l'application, nous utiliserons donc une seule et même instance du serveur GlassFish pour héberger la totalité des composants.

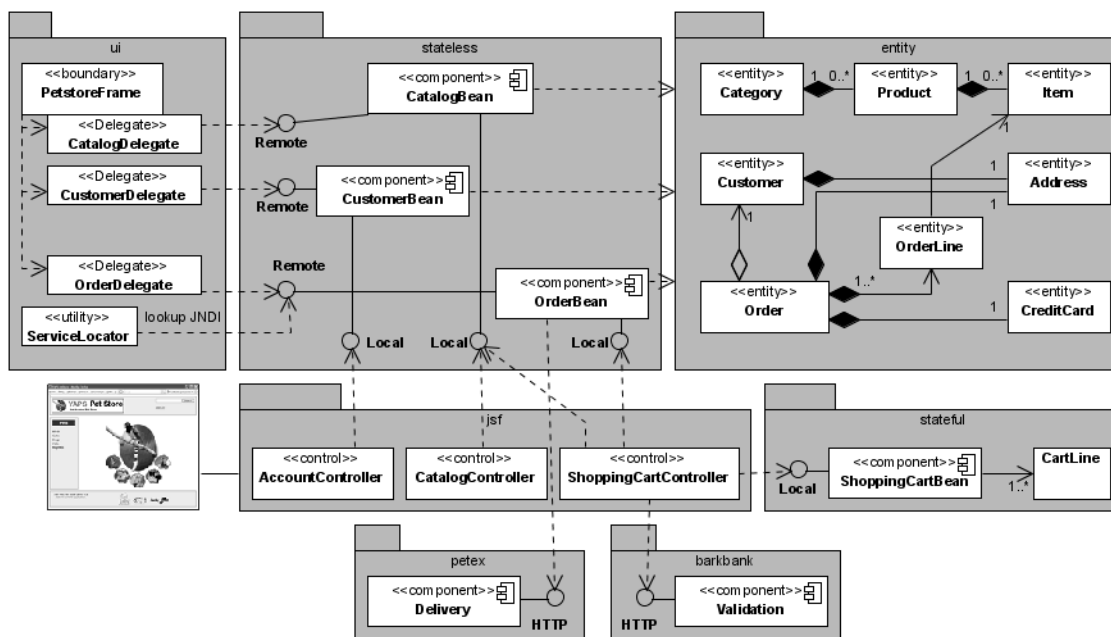


Figure 9–12 Services web dans l'architecture YAPS Pet store

Compiler

Pour compiler les classes qui se trouvent dans les répertoires `src` et générer les artefacts côté serveur, on utilise les tâches Ant `barkbank-compile` et `petex-compile`. Ces tâches se chargent de compiler les classes et de les placer dans les répertoires `classes` de chaque projet. Pour supprimer tous les répertoires de travail (`build` et `classes`), on peut utiliser les tâches `barkbank-clean` et `petex-clean`.

ANT Les tâches dans build.xml et admin.xml

Les fichiers contenant les tâches Ant (`build.xml` et `admin.xml`) sont décrits en annexe et téléchargeables sur le site :

► <http://www.antoniogoncalves.org>

Packager

Les services de vérification de la BarkBank et de transport PetEx sont chacun packagés dans une application web. Les archives contiennent les classes des services web mais aussi tous les artefacts serveurs générés par `wsgen` (les classes et les fichiers WSDL et XSD). Pour ce faire, on exécute les tâches `barkbank-build` et `petex-build`. Les archives `barkbank.war` et `petex.war` sont placés dans le répertoire `build`.

Figure 9–13
Contenu de l'archive
barkbank.war



Figure 9–14
Contenu de l'archive
petex.war



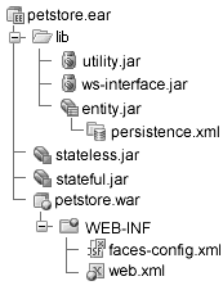


Figure 9-15 Contenu du fichier petstore.ear

ANT Deploy, undeploy

Une fois les applications déployées à l'aide des tâches `deploy`, on peut les supprimer du serveur GlassFish en utilisant les tâches suivantes :

- `yaps-undeploy` ;
- `barkbank-undeploy` ;
- `petex-undeploy`.

Pour générer les artefacts client pour l'application YAPS Pet Store, il est impératif que les services web soient déployés. Ensuite, il suffit d'exécuter la tâche `yaps-build` qui se chargera de générer les artefacts (avec `wsimport`) et de packager la totalité des classes dans le fichier `petstore.ear`. Les artefacts des deux services web se trouvent dans le fichier `ws-interface.jar` qui se trouve dans le sous-répertoire `lib`.

Déployer

Comme toujours pour le déploiement, il faut s'assurer que le serveur GlassFish et la base de données Derby soient démarrés. Si ce n'est pas le cas, utilisez les tâches Ant d'administration :

```
%PETSTORE_HOME%\> ant -f admin.xml start-domain
%PETSTORE_HOME%\> ant -f admin.xml start-db
```

Chaque application est packagée dans un fichier d'archive différent et doit être déployée séparément. Pour cela, utilisez les tâches Ant `petex-deploy`, `barkbank-deploy` et `yaps-deploy`. Vous pouvez ensuite accéder individuellement à chacune d'elle en utilisant des URL différentes :

- `http://localhost:8080/petstore`
- `http://localhost:8080/barkbank`
- `http://localhost:8080/petex`

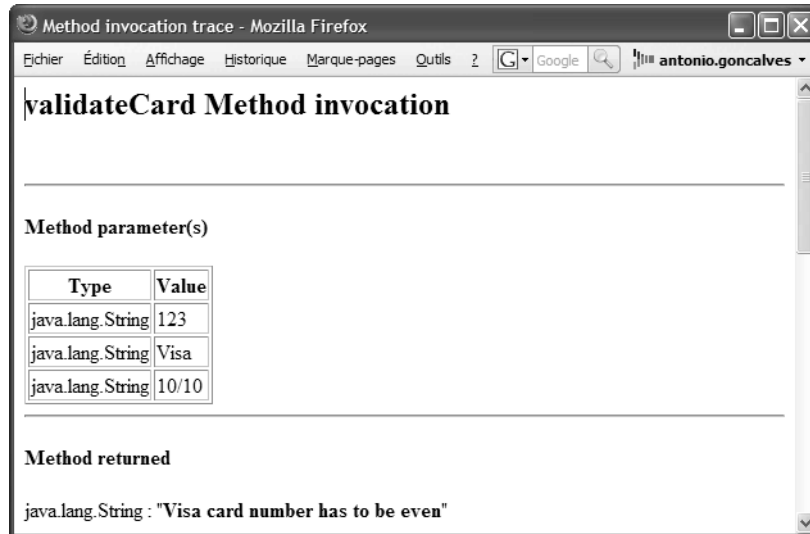
Tester les services web avec GlassFish

Une fois les applications déployées, vous pouvez utiliser le mécanisme de GlassFish pour tester les services web. Par exemple, pour tester le service de validation de BarkBank, il suffit de vous rendre à l'adresse suivante : `http://localhost:8080/barkbank/ValidationService?Tester`.



Figure 9-16
Page permettant de tester
le service de validation

Il est alors possible de saisir les paramètres que l'on souhaite et invoquer le service web. Par exemple, dans l'écran précédent (figure 9-16), vous pouvez saisir un numéro impair pour une carte Visa. Lorsque vous cliquez sur le bouton `validateCard`, le service est invoqué et le résultat s'affiche dans la page ci-après (figure 9-17), vous indiquant que le numéro de la carte est erroné car impair (*Visa card number has to be even*).



REMARQUE Le message Soap

La page de résultat du service web affiche aussi la requête et la réponse de la méthode au format Soap. Cela permet de voir ce qui est créé automatiquement sans que nous n'ayons eu à manipuler les messages Soap directement.

Figure 9-17
Page de résultat du service web

Pour consulter le contenu du document WSDL vous pouvez aussi vous rendre à l'adresse suivante :

<http://localhost:8080/barkbank/ValidationService?WSDL>.

Exécuter

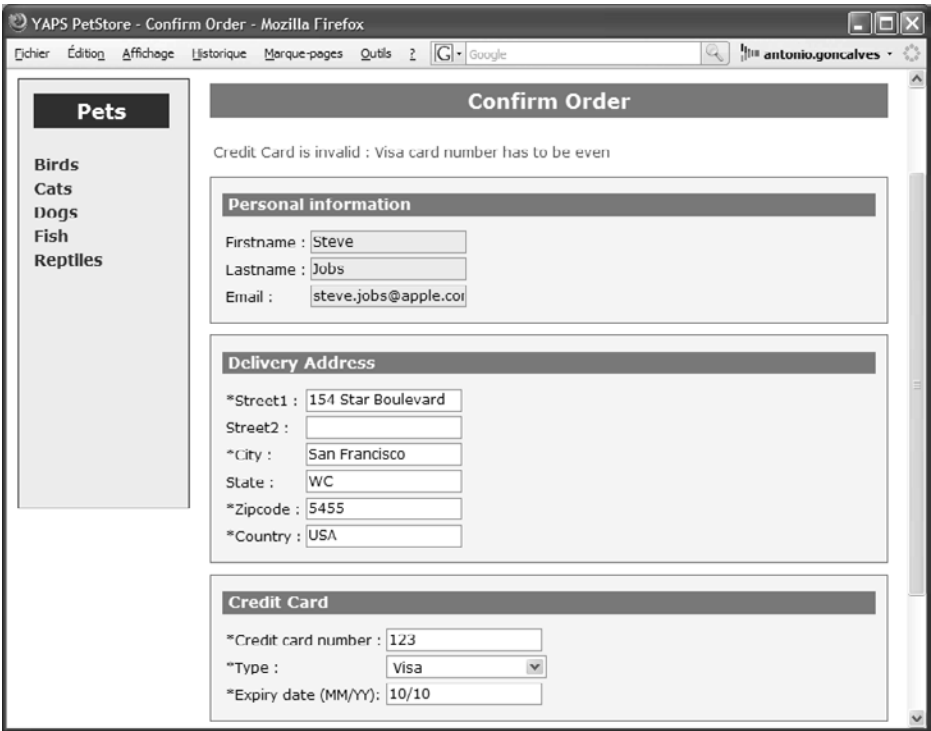
Pour vérifier que tout cela fonctionne, il suffit d'utiliser l'application YAPS Pet Store pour acheter des articles. Au moment de valider le panier électronique, vous pouvez saisir un mauvais numéro de carte bancaire, par exemple, et vérifier que le statut s'affiche à l'écran (figure 9-18).

Saisissez maintenant des données correctes. Cela a pour effet de créer un bon de commande et d'avertir le transporteur PetEx via son service web. Pour vérifier que le service a bel et bien été invoqué, consultez les logs du serveur GlassFish. Vous pouvez ainsi voir l'appel au service `Delivery`.

GLASSFISH Consulter les logs

Pour lire les logs du serveur GlassFish vous pouvez, soit consulter le fichier `%GLASSFISH_HOME%\domains\petstore\logs`, soit vous connecter à la console d'administration. Pour cela, allez à l'adresse `http://localhost:8282` puis saisissez le nom de l'utilisateur `admin` et son mot de passe `adminpwd`. Cliquez sur le menu *Application Server -> View Log Files*.

Figure 9–18
Un message s’affiche indiquant
que le numéro est invalide.



Extrait des logs du serveur GlassFish

```
com.petex.transport|Delivery Order Received

com.petex.transport|Deliver from DeliveryPlace{contact='Yaps
Inc.', street='125, Poodle Square', city='San Francisco',
state='LA', zipcode='16354', country='USA'}

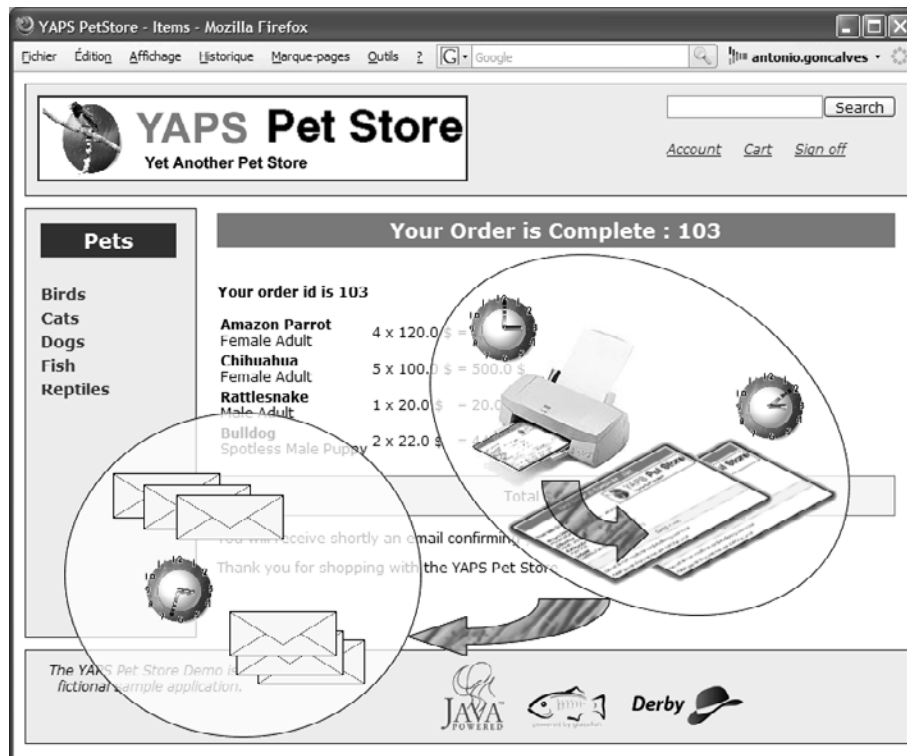
com.petex.transport|Deliver to DeliveryPlace{contact='Jobs',
street='154 Star Boulevard', city='San Francisco', state='WC',
zipcode='5455', country='USA'}

com.petex.transport|Reference n° 101
```

En résumé

Les clients peuvent maintenant acheter des animaux en ligne. Notre application a donc besoin de communiquer avec des systèmes externes pour vérifier la validité des cartes bancaires ainsi que pour avertir le transporteur de nouvelles livraisons. Ces traitements B2B se font via l'échange de données au format XML. Ce chapitre nous a présenté les services web et les différentes API sous-jacentes à cette technologie. Java EE 5 et la génération d'artefacts simplifient grandement le développement des web services.

chapitre 10



Traitements asynchrones

Lorsque le bon de commande est créé, le système doit l'imprimer et envoyer au client un e-mail lui confirmant ses achats. Ces traitements pouvant être longs, ils sont effectués de manière asynchrone via un système d'échange de messages. Ce chapitre nous présente les technologies JMS, MDB ainsi que l'API JavaMail.

SOMMAIRE

- ▶ Traitements asynchrones
- ▶ Envoi d'un e-mail de confirmation
- ▶ Impression du bon de commande
- ▶ JMS et les message-driven beans
- ▶ Cycle de vie des MDB

MOTS-CLÉS

- ▶ JMS
- ▶ MOM
- ▶ MDB
- ▶ Point à point
- ▶ Publication/abonnement
- ▶ JavaMail

⚡ Synchrone/Asynchrone

- Synchrone : échange ou traitement d'informations en direct (appel bloquant).
- Asynchrone : échange ou traitement d'informations en différé.

RETOUR D'EXPÉRIENCE Les Threads dans Java EE

Pour effectuer un traitement asynchrone en Java, on peut utiliser l'API des Threads. En effet, il suffit à une classe serveur de créer un nouveau Thread par traitement demandé. Cette API s'est enrichie et simplifiée avec Java SE 5. Malheureusement, les spécifications Java EE interdisent l'utilisation de Threads dans les EJB car ce travail doit être fait par le conteneur et non par le développeur. Il est donc proscrit d'utiliser les Threads dans un environnement Java EE.

APPROFONDIR JMS

- 📖 Eric Bruno, *Java Messaging*, Charles River, 2005
- 📖 Richard Monson-Haefel, *Java Message Service*, O'Reilly, 2002
- 📖 Kareem Yusuf, *Enterprise Messaging Using JMS and IBM Websphere*, IBM Press, 2004
- ▶ <http://java.sun.com/products/jms/>

OUTILS Providers JMS

- IBM WebSphere MQ
 - ▶ <http://www-306.ibm.com/software/integration/wmq/>
- JBoss MQ (bientôt remplacé par JBoss Messaging dans JBoss AS 5.0)
 - ▶ <http://www.jboss.org/wiki/Wiki.jsp?page=JBossMQ>
- Sonic MQ
 - ▶ <http://www.sonicsoftware.com/products/sonicmq>
- Sun OpenMQ
 - ▶ http://www.sun.com/software/products/message_queue/index.xml

Lorsque le client valide son panier électronique et qu'il saisit son adresse de livraison ainsi que ses coordonnées bancaires, le système crée un nouveau bon de commande. Au même moment, il envoie un e-mail de confirmation au client lui détaillant le contenu de sa commande, et imprime le bon de commande pour pouvoir être archivé par le service comptable.

L'impression et l'envoi de l'e-mail peuvent s'avérer être des traitements longs. Imaginez que l'imprimante soit débranchée, qu'elle manque de papier ou que l'adresse e-mail soit erronée ou le pare-feu en panne. Pour toutes ces raisons, il est préférable d'effectuer ces traitements de manière asynchrone. La création du bon de commande dans le système peut se faire sans attendre que l'impression soit effectuée.

Dans le même esprit, les employés de YAPS doivent être avertis en temps réel de la création d'un bon de commande contenant des reptiles. Ces alertes se font par envoi de messages et sont affichées sur l'IHM des employés. Si les employés ne sont pas connectés à leur application, les alertes doivent être empilées jusqu'à ce qu'ils se reconnectent.

Ce chapitre couvre les fonctionnalités restantes du cas d'utilisation « Créer un bon de commande ».

JMS

JMS, ou *Java Message Service*, est une API d'échange de messages pour permettre un dialogue entre applications via des brokers de messages ou MOM (*Middleware-Oriented Messages*). L'application cliente envoie un message dans une file d'attente (plutôt qu'à une application, ce qui permet de faire du découplage logiciel), sans se soucier de la disponibilité de cette application (chaque système possède son propre cycle de vie). Le client a, de la part du broker de messages, une garantie de qualité de service (certitude de remise au destinataire, délai de remise, etc.).

L'API JMS, contenue dans le paquetage `javax.jms`, définit plusieurs entités :

- **Un provider** : outil (broker de messages) qui implémente l'API JMS pour échanger les messages entre deux clients. Le serveur GlassFish utilise l'implémentation Sun Java System Message Queue (OpenMQ).
- **Un client** : classe Java qui utilise JMS pour émettre et/ou recevoir des messages. Un client envoie un message vers une file d'attente, et le client destinataire reste à l'écoute d'une file d'attente pour recevoir le message. Le transfert du message et sa persistance sont assurés par le provider.

- **Un message** : données échangées de manière asynchrone entre les composants. Il existe plusieurs types de messages (texte, objet, binaire, etc.).
- **Les objets administrés** : ressources à rechercher dans l'annuaire JNDI du provider, telles que les fabriques de connexions et les destinations.

Les messages

Pour dialoguer, les clients JMS s'échangent des messages, c'est-à-dire qu'un client expédie un message vers une file d'attente, et qu'un client destinataire exécute un traitement à la réception de ce message. En JMS, ces messages doivent implémenter l'interface `javax.jms.Message` et sont composés de trois parties :

- **l'en-tête (header)**, qui comporte des caractéristiques techniques (identifiant, date d'envoi, etc.) ;
- **les propriétés (properties)**, qui représentent les caractéristiques fonctionnelles du message ;
- **le corps du message (body)**, qui contient les données à transporter.

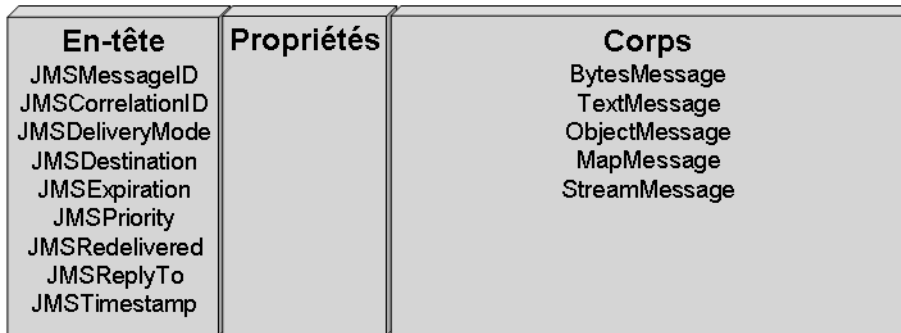


Figure 10-1
Anatomie d'un message JMS

L'en-tête du message

L'en-tête du message contient un certain nombre de champs prédéfinis permettant de l'identifier. On peut voir cette section comme les métadonnées du message : qui a créé le message, date de création, durée de vie, accusé de réception demandé ou non, etc. Chacune de ces métadonnées possède des accesseurs (définis dans l'interface `javax.jms.Message`) qui permettent d'en modifier le contenu, mais la plupart sont affectées automatiquement par le provider.

Tableau 10-1 Métadonnées de l'en-tête

Nom	Description
JMSMessageID	Identifiant unique du message.
JMSCorrelationID	Utilisé pour associer de façon applicative deux messages par leur identifiant.
JMSDeliveryMode	Il existe deux modes d'envoi : persistant (le message est délivré une et une seule fois au destinataire, c'est-à-dire que même en cas de panne du provider, le message sera délivré) et non persistant (le message peut ne pas être délivré en cas de panne puisqu'il n'est pas persisté).
JMSDestination	File d'attente destinataire du message.
JMSExpiration	Date d'expiration du message.
JMSPriority	Priorité du message. Cet attribut numérique indique la priorité de façon croissante à partir de 0 (les messages de niveau 9 ont une priorité plus importante que les messages de niveau 0).
JMSRedelivered	Booléen qui signifie que le message a été redélivré au destinataire.
JMSReplyTo	File d'attente de réponse du message.
JMSTimestamp	L'heure d'envoi du message est affectée automatiquement par le provider.

Les propriétés

Cette section du message est optionnelle et agit comme une extension des champs d'en-tête. Les propriétés d'un message JMS sont des couples (nom, valeur), où la valeur est un type de base du langage Java (entiers, chaînes de caractères, booléens, etc.). L'interface `javax.jms.Message` définit des accesseurs pour manipuler ces valeurs. Ces données sont généralement positionnées par le client avant l'envoi d'un message et, comme nous le verrons par la suite, peuvent être utilisées pour filtrer les messages.

Le corps du message

Le corps du message, bien qu'optionnel, est la zone qui contient les données. Ces données sont formatées selon le type du message qui est défini par les interfaces suivantes :

Tableau 10-2 Types de messages

Interface	Description
<code>javax.jms.BytesMessage</code>	Pour les messages binaires (images, vidéos, documents électroniques...).
<code>javax.jms.TextMessage</code>	Échange de données de type texte (XML par exemple).
<code>javax.jms.ObjectMessage</code>	Messages composés d'objets Java sérialisés.
<code>javax.jms.MapMessage</code>	Échange de données sous la forme clé/valeur. La clé doit être une String et la valeur de type primitif.
<code>javax.jms.StreamMessage</code>	Échange de données en provenance d'un flux.

Il est possible de définir son propre type de message en implémentant l'interface mère `javax.jms.Message`. Lors de la réception d'un message, celui-ci est toujours de type `javax.jms.Message` ❶. Il doit donc être trans-

typé ❸ en fonction de son type. L'opérateur `instanceof` ❷ est alors utilisé pour détecter le type exact du message. À ce moment, il faut utiliser le getter ❹ correspondant pour obtenir les données (`ObjectMessage.getObject()`, `TextMessage.getText()`, etc.).

Exemple de transtypage d'un message

```
public void onMessage(javax.jms.Message ❶ message) {
    if (message instanceof ❷ ObjectMessage) {
        ObjectMessage objMsg = (ObjectMessage) message; ❸
        objMsg.getObject(); ❹
    }
}
```

Les objets administrés

On nomme ces objets « administrés » car ils doivent être auparavant créés et configurés de manière administrative, via une console ou par script.

Dans l'utilisation de l'API JMS, deux types d'objets sont gérés différemment des autres :

- Les fabriques de connexions (`ConnectionFactory`) permettent d'obtenir une connexion auprès d'un provider.
- Les destinations sont les objets qui véhiculent les messages. JMS comporte deux types de destinations, les `Queue` et les `Topic`.

Pour obtenir une `ConnectionFactory`, une `Queue`, ou un `Topic`, il faut les rechercher par leur nom dans l'annuaire JNDI ou utiliser l'injection. Cela suppose donc que ces objets aient été préalablement inscrits dans JNDI. C'est ce que nous avons fait dans le chapitre 3, *Outils et installation*, lorsque nous avons configuré GlassFish.

REMARQUE La fabrique et la destination JMS

Lors de la configuration du serveur GlassFish, nous avons créé une fabrique de connexions (`javax/petstoreConnectionFactory`) ainsi qu'une file d'attente (`javax/topic/order`).

La fabrique de connexions

Dans JMS, la fabrique de connexions (`ConnectionFactory`) permet d'obtenir une connexion au provider (Sun Java System Message Queue dans le cas de GlassFish). Une fois la connexion obtenue, on peut envoyer ou recevoir des messages. Comme nous le verrons plus bas, l'objet `ConnectionFactory` permet d'obtenir les objets `Connection`, `Session` puis les objets pour produire (`MessageProducer`) et consommer (`MessageConsumer`) des messages.

Dans les chapitres précédents, nous avons utilisé l'annotation `@EJB` pour injecter les références de nos stateless session beans. Cette annotation est propre aux EJB. Pour la fabrique de connexions, nous utiliserons l'annotation plus générique `@javax.annotation.Resource`.

ANNOTATION **L'injection avec @Resource**

L'annotation `@Resource` permet d'injecter la référence d'une ressource dans un attribut de classe. Dans nos exemples, nous l'utilisons pour référencer des destinations ou fabriques de connexions JMS. Mais la même annotation peut être utilisée pour référencer une `DataSource`, un pool de connexions JDBC, ou tout autre objet déclaré dans JNDI.

Cette annotation s'applique à une classe, une méthode ou un attribut.	➤
Nom de la ressource JNDI.	➤
Classe de la ressource (ex. <code>javax.sql.DataSource</code> pour une source de données).	➤
Informe le serveur du composant responsable de l'authentification pour accéder à la ressource : le conteneur ou l'application.	➤
Ressource partageable ou non.	➤
Cet attribut représente le nom donné à la ressource à l'intérieur du conteneur. Il est spécifique à chaque serveur d'applications et peut donc ne pas être portable.	➤
Description de la ressource.	➤

Injection de la fabrique de connexions

```
@Resource(mappedName = "jms/petstoreConnectionFactory")
private ConnectionFactory connectionFactory;
```

`@Resource` permet d'injecter des ressources externes, dans notre cas la fabrique de connexions.

Code de l'annotation `@javax.annotation.Resource`

```
package javax.annotation;

@Target({TYPE, FIELD, METHOD}) @Retention(RUNTIME)

public @interface Resource {

    String name() default "";

    Class type() default Object.class

    AuthenticationType authenticationType() default CONTAINER;

    boolean shareable() default true;

    String mappedName() default "";

    String description() default "";

}
```

Destinations

JMS définit deux types de destinations correspondant aux deux modes d'envoi des messages :

- **Le mode point à point (Point to Point)** utilise des files d'attente (`javax.jms.Queue`) pour communiquer. Ce mode (un émetteur, un récepteur) s'apparente à l'envoi d'un e-mail.
- **Le mode publication/abonnement (Publish/Subscribe)** utilise des sujets (`javax.jms.Topic`) pour échanger des messages. Ce mode (un émetteur, multiples récepteurs) correspond, par exemple, à une souscription auprès d'un serveur de news. Par défaut, seuls les récepteurs connectés au `Topic` sont alertés de l'arrivée du message. Pour que les messages soient conservés pour les récepteurs déconnectés, ils doivent avoir été déclarés comme durables.

Chaque mode utilise une interface différente pour envoyer des messages : `javax.jms.QueueSender` dans le mode point à point, et `javax.jms.TopicPublisher` pour le mode publication/abonnement. Toutes deux héritent de la super interface `javax.jms.MessageProducer`.

Pour référencer un Topic, par exemple, on utilise le système d'injection de l'annotation `@Resource`.

Injection de la destination

```
@Resource(mappedName = "jms/topic/order")
private Topic destinationOrder;
```

Le mode point à point

Le mode point à point repose sur le concept de files d'attente (Queue). Cela signifie que chaque message est envoyé par un producteur dans une file d'attente, et est reçu par un seul consommateur. Une fois le message consommé, il disparaît de la file d'attente.

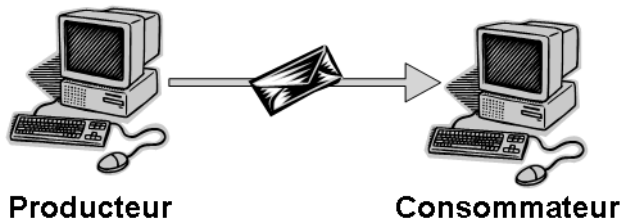


Figure 10–2
Mode point à point

Bien entendu, tant que le message n'est pas consommé, ou qu'il n'a pas expiré, il reste stocké au sein du provider. Le client peut donc le consommer ultérieurement.

Il peut arriver aussi que le consommateur du message l'ait reçu par erreur, ou qu'il ne puisse pas le traiter immédiatement. Dans ce cas, il peut annuler la réception du message en effectuant un `rollback`.

Le mode publication/abonnement

Le mode publication/abonnement repose sur le concept de sujets (Topics). Cela signifie que des messages sont envoyés par plusieurs producteurs dans un Topic, et qu'ils sont reçus par plusieurs consommateurs. Les consommateurs des messages s'abonnent aux sujets qui les intéressent : c'est le principe de l'abonnement. L'émetteur du message ne connaît pas les destinataires qui se sont abonnés.

Contrairement au mode point à point, dans le mode publication/abonnement un message envoyé va être reçu par plusieurs clients.

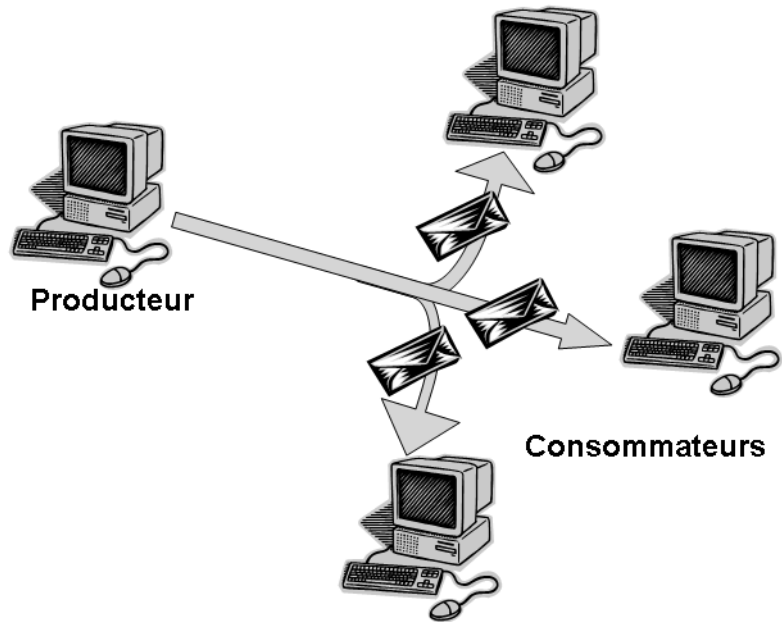


Figure 10-3
Mode publication/abonnement

Envoyer les messages

Voyons maintenant comment utiliser tous ces concepts pour publier un message dans un Topic. Comme nous l'avons vu, la fabrique de connexions et la destination doivent être connues par les clients JMS. Pour cela, nous utilisons l'annotation `@javax.annotation.Resource` qui permet d'injecter la référence de la fabrique JMS que nous avons créée dans GlassFish ❶, ainsi que la destination `jms/topic/order` ❷.

Une fois la référence de la `ConnectionFactory` obtenue, on se connecte au provider JMS via l'objet `javax.jms.Connection` ❸. À partir de cette connexion, on obtient une session ❹. Une Session est un contexte transactionnel utilisé pour grouper un ensemble d'envois de messages (ou de réceptions de messages) dans une unité de travail. Comme avec les bases de données, une session transactionnelle n'est validée qu'après appel implicite ou explicite d'un ordre commit.

À partir de la session, on crée un `MessageProducer` ❺ qui va permettre d'envoyer ❷ un message auprès d'une destination. La session permet aussi de créer le message ❻ (de type objet dans notre exemple).

TRANSACTION JMS

Dans le cas d'une session transactionnelle (`connection.createSession(true)`), l'envoi de messages n'est effectivement réalisé qu'au moment de l'exécution du `commit` de la transaction.

Envoi d'un message

```
public class OrderBean implements OrderRemote, OrderLocal {
    @Resource(mappedName = "jms/petstoreConnectionFactory") ❶
    private ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/topic/order") ❷
    private Topic destinationOrder;
    private void publishOrder(Order order) {
        Connection connection = ❸
            connectionFactory.createConnection();
        Session session = connection.createSession(true,
            Session.AUTO_ACKNOWLEDGE); ❹
        MessageProducer producer =
            session.createProducer(destinationOrder); ❺
        ObjectMessage objectMessage =
            session.createObjectMessage(); ❻
        objectMessage.setObject(order);
        producer.send(objectMessage); ❼
        session.close();
        connection.close();
    }
}
```

❶ Injection de la fabrique de connexions et du Topic qui sont déclarés dans GlassFish.

❷ On se connecte au provider de messages.

❸ Le premier paramètre true signifie que la session est transactionnelle.

❹ On crée un MessageProducer, qui dans notre cas sera un TopicPublisher.

❺ On crée un message de type objet. On affecte au corps du message l'entity bon de commande.

❻ Le message est envoyé dans le Topic.

❼ On referme la session et la connexion.

Recevoir un message

Le consommateur du message est le client capable d'être à l'écoute d'une file d'attente (ou d'un Topic), et de traiter les messages à leur réception. Si on ne veut pas bloquer le consommateur, le mécanisme d'écoute doit être fait dans un Thread à part. En effet, le client doit être constamment à l'écoute (listener) et, à l'arrivée d'un nouveau message, il doit pouvoir le traiter. Pour cela, le Thread doit appeler la méthode `onMessage` de l'interface `javax.jms.MessageListener`. Celle-ci permet la réception asynchrone des messages et ne dispose que d'une méthode dont la signature est la suivante :

```
public void onMessage(javax.jms.Message message);
```

Charge au développeur d'implémenter cette interface pour réaliser le traitement adéquat lors de la réception d'un message.

Prenons l'exemple du client Swing de l'application. Celui-ci doit être à l'écoute des nouvelles commandes qui sont publiées dans le Topic "jms/topic/order". Lorsque le message contenant le nouveau bon de commande arrive, ses informations sont affichées à l'écran.

ANNOTATION @Resource dans le client Swing

Souvenez-vous que pour des raisons pédagogiques l'interface Swing ne s'exécute pas dans un conteneur client (*Application Client Container*). Si tel avait été le cas, nous aurions pu utiliser l'annotation `@Resource` pour injecter les références de la fabrique de connexions et de la destination JMS et ainsi s'affranchir de JNDI.

JMS Acquittement de message

Lorsqu'on crée une session, on peut spécifier le mode d'acquittement des messages. Il en existe trois :

- acquittement automatique (`Session.AUTO_ACKNOWLEDGE`);
- acquittement fait par le client (`Session.CLIENT_ACKNOWLEDGE`);
- duplication des acquittements tolérée (`Session.DUPS_OK_ACKNOWLEDGE`).

L'injection n'est pas utilisée.

On recherche la fabrique de connexions et le Topic déclarés dans l'annuaire JNDI de l'instance Glassfish.

On se connecte au provider de messages.

Le premier paramètre `true` signifie que la Session est transactionnelle.

On crée un `MessageProducer`, qui dans notre cas sera un `TopicPublisher`.

On associe un listener à la classe.

On démarre la connexion.

Les messages arrivent par la méthode `onMessage`.

Pour effectuer cette tâche, le client a tout d'abord besoin d'implémenter l'interface `MessageListener` ❶. Il doit ensuite obtenir la fabrique de connexions et la destination JMS ❷ sur lesquelles il souhaite écouter l'arrivée des nouveaux messages. L'application Swing s'exécutant en dehors d'un conteneur, il nous faut utiliser l'API JNDI pour obtenir les références ❸.

Tout comme pour l'envoi d'un message, le consommateur doit se connecter au provider via l'objet `javax.jms.Connection` ❹ pour en retour obtenir une session ❺. À partir de la session, on crée un `MessageConsumer` ❻ qui va permettre de consommer les messages. Pour ce faire, on associe un listener pour traiter les messages de façon asynchrone ❼. Ainsi, à chaque réception d'un nouveau message, la méthode `onMessage` est automatiquement invoquée ❽ et peut effectuer un traitement.

Réception d'un message

```
public class YapsMsgListener implements MessageListener {❶

    private ConnectionFactory connectionFactory;❷
    private Topic destinationOrder;

    private void receiveOrder(Order order) {
        // Pour simplifier la lecture du code, la réception
        // asynchrone à l'aide d'un thread n'est pas implémentée

        InitialContext ic = new InitialContext();❸
        connectionFactory = (ConnectionFactory)
            ic.lookup("jms/petstoreConnectionFactory");
        destinationOrder = (Topic)
            ic.lookup("jms/topic/order");

        Connection connection =❹
            connectionFactory.createConnection();

        Session session = connection.createSession(true,
            Session.AUTO_ACKNOWLEDGE);❺

        MessageConsumer consumer =
            session.createConsumer(destinationOrder);❻

        consumer.setMessageListener(this);❼

        connection.start();

    }

    public void onMessage(Message message) {❽
        if (message instanceof ObjectMessage) {
            ObjectMessage objMsg = (ObjectMessage) message;
            tableModel.add(objMsg.getObject());
        }
    }
}
```

La sélection de messages

Pour les clients qui ne seraient pas intéressés par tous les messages arrivant dans une file d'attente, JMS permet de les filtrer. Ce filtrage est effectué par le provider de messages plutôt que par l'application elle-même. Ainsi, selon certains critères, le client ne recevra que les messages qui l'intéressent. Les critères de sélection ne peuvent porter que sur des champs inclus dans l'en-tête ou dans les propriétés du message. Il n'est pas possible d'utiliser les données du corps pour effectuer le filtre.

Cette possibilité de filtrer les messages se fait par le consommateur qui utilise un sélecteur de type `SQL`. Cette chaîne de caractères permet de ne sélectionner que les messages dont les champs d'en-tête et les propriétés présentent certaines caractéristiques.

Exemple de filtres JMS

```
"JMSPriority > 6"
```

```
"JMSPriority > 6 And OrderId < 100"
```

```
"JMSPriority > 6 And OrderId < 100 Or Reptiles=true"
```

Reprenons l'exemple de l'interface Swing. Les employés ne sont en réalité intéressés que par les commandes contenant des reptiles. Pour cela, le producteur du message rajoute une propriété `Reptiles` ❶ de type booléen qu'il positionne à « vrai » lorsque qu'une commande contient au moins un reptile.

Le producteur du message rajoute la propriété `Reptiles`

```
ObjectMessage objectMessage = session.createObjectMessage();
objectMessage.setObject(order);
```

```
objectMessage.setBooleanProperty("Reptiles", true); ❶
```

```
producer.send(objectMessage);
```

Le consommateur, lui, utilise un sélecteur ❷ pour ne recevoir que les messages possédant une propriété ayant le nom de `Reptiles` avec une valeur à `true`. La session permet de préciser dans ces paramètres une chaîne de caractères qui va servir de filtre sur les messages à recevoir.

Le consommateur utilise un sélecteur sur les propriétés

```
Session session = connection.createSession(true,
                                           Session.AUTO_ACKNOWLEDGE);
```

```
MessageConsumer consumer = ❷
session.createConsumer(destinationOrder, "Reptiles=true"); ❷
consumer.setMessageListener(this);
```

JMS Sélection sur l'en-tête

JMS permet de faire une sélection sur les champs de l'en-tête `JMSDeliveryMode`, `JMSPriority`, `JMSMessageID`, `JMSCorrelationID`, `JMSType` et `JMSTimestamp`. Celle-ci n'est pas possible pour `JMSDestination`, `JMSReplyTo`, `JMSExpiration` ou `JMSRedelivered`.

- ◀ Reçoit les messages de priorité supérieurs à 6. `JMSPriority` est un champ de l'en-tête.
- ◀ Reçoit les messages de priorité supérieurs à 6 et ayant une propriété `OrderId` inférieure à 100...
- ◀ ... ou une propriété `Reptiles` à vrai.

- ◀ Le producteur crée un message de type objet et affecte un bon de commande dans le corps du message.
- ◀ On rajoute une propriété de type booléen sur laquelle le filtrage s'effectuera.
- ◀ Le message est envoyé.

- ◀ Le consommateur ne recevra que les messages dont la propriété `Reptiles` est « vrai ».

APPROFONDIR **MDB**

- 📖 Ed Roman, Rima Patel Sriganesh, Gerald Brose, *Mastering Enterprise JavaBeans, 3rd Edition*, Wiley, 2004
- <http://java.sun.com/javaee/5/docs/tutorial/doc/>

SYNTAXE **Classe EJB**

La classe de l'EJB doit suivre les règles de développement suivantes :

- elle doit être annotée par `@MessageDriven` ;
- elle doit être publique ;
- elle ne doit pas être finale ;
- elle ne doit pas être abstraite ;
- elle ne doit pas définir la méthode `finalize` ;
- elle doit avoir un constructeur par défaut ;
- elle doit implémenter les méthodes de ses interfaces.

Un MDB est défini comme tel grâce à l'annotation `@MessageDriven`.

Point d'entrée du MDB.

On transtype le message en `ObjectMessage`.

On récupère le bon de commande qui se trouve dans le corps du message.

Message-driven bean

Un *message-driven bean*, ou MDB, est un EJB qui se comporte comme un listener JMS, c'est-à-dire qui reçoit des messages et les traite de manière asynchrone. Les MDB se rapprochent des EJB stateless car ils sont, eux aussi, sans état. Ils s'exécutent à l'intérieur d'un conteneur qui assure le multithreading, la sécurité ou la gestion des transactions.

Les MDB sont à l'écoute (listener) d'une file d'attente et se réveillent à chaque arrivée de messages. En fait, il faut garder à l'esprit que c'est le conteneur qui est le véritable listener JMS et qu'il délègue au MDB le traitement du message, et plus particulièrement à la méthode `onMessage()`. Comme les autres EJB, le MDB peut accéder à tout type de ressources : EJB, JDBC, mail, etc.

Exemple de message-driven bean

Un MDB ne possède pas d'interface distante ou locale puisqu'il n'est pas utilisé par un client. Il est constitué d'une seule classe Java qui doit être annotée par `@javax.ejb.MessageDriven` ❶. Pour réagir à l'arrivée d'un message, il doit implémenter la méthode `onMessage(javax.jms.Message)` ❸ définie dans l'interface `javax.jms.MessageListener` ❷. Il est associé à une destination JMS, c'est-à-dire une Queue pour les communications point à point ou à un Topic pour les communications publication/souscription. La méthode `onMessage` est activée à la réception d'un message envoyé par un client JMS.

Classe du message-driven bean

```
@MessageDriven(mappedName = "jms/topic/order") ❶

public class OrderPrinterBean implements MessageListener ❷{

    public void onMessage(Message message) { ❸

        if (message instanceof ObjectMessage) {
            ObjectMessage msg = (ObjectMessage) message;
            Order order = (Order) msg.getObject();

            printOrder(order);
        }
        // gestion des exceptions
    }
}
```

Comme la plupart des composants spécifiés dans Java EE 5, les message-driven beans utilisent, eux aussi, les annotations.

Code de l'annotation @javax.ejb.MessageDriven

```
package javax.ejb;

@Target({TYPE}) @Retention(RUNTIME)
public @interface MessageDriven {

    String name() default "";

    Class messageListenerInterface() default Object.class;

    ActivationConfigProperty[] activationConfig() default {};

    String mappedName() default "";

    String description() default "";
}
```

- ◀ Cette annotation s'applique à une classe.
- ◀ Nom du MDB.
- ◀ Le type d'interface implémentée par le MDB peut être défini par cet attribut (MessageListener dans notre exemple).
- ◀ Permet de configurer le MDB.
- ◀ Nom JNDI de la destination sur laquelle le MDB est à l'écoute.
- ◀ Description du message-driven bean.

Les MDB peuvent être à l'écoute de messages arrivant de différents types de providers. Pour cela, l'annotation @javax.ejb.MessageDriven permet au MDB de configurer certains paramètres de ses providers. Il suffit d'utiliser le tableau activationConfig comme suit :

Exemple de configuration de MDB

```
@MessageDriven(mappedName = "jms/topic/order",
    activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationType",
            ❶ propertyValue = "javax.jms.Topic"),
        @ActivationConfigProperty(propertyName = "subscriptionDurability",
            ❷ propertyValue = "Durable"),
        @ActivationConfigProperty(propertyName = "subscriptionName",
            propertyValue = "EmailSender")
    }
)
public class EmailSenderBean implements MessageListener {
    (...)
}
```

Dans cet exemple, on utilise l'annotation @ActivationConfigProperty pour spécifier le type de la destination ❶ ou la persistance des messages ❷. @javax.ejb.ActivationConfigProperty est constituée de clés et de valeurs permettant une configuration plus fine d'un MDB.

Code de l'annotation @javax.ejb.ActivationConfigProperty

```
package javax.ejb;

@Target({}) @Retention(RUNTIME)
```

- ◀ Cette annotation s'applique à @MessageDriven.

MDB Implémenter MessageListener ?

La spécification Java EE 5 est assez vague sur l'obligation d'implémenter l'interface MessageListener ou non. Dans un chapitre, on nous dit qu'il n'est pas nécessaire pour un MDB de l'implémenter, et dans un autre, on nous dit presque le contraire. Avec l'implémentation GlassFish, pour qu'un MDB soit reconnu comme tel, il lui faut juste utiliser l'annotation @MessageDriven. Pas besoin d'implémenter une quelconque interface. J'ai néanmoins laissé cette interface dans les exemples de code de ce chapitre.

Clé de la propriété.	▶
Valeur de la propriété.	▶

```
public @interface ActivationConfigProperty {
    String propertyName();
    String propertyValue();
}
```

RETOUR D'EXPÉRIENCE Intéropérabilité avec JMS

Dans le précédent chapitre, *Échanges B2B*, nous avons vu les services web comme outils d'interopérabilité avec des systèmes externes. On peut aussi utiliser les MOM (*Middleware Oriented Messages*) pour le faire. Imaginez une application externe développée dans un langage différent de Java (C++, PHP, .Net, etc.) et s'exécutant sur un système d'exploitation autre que le vôtre. Il suffit qu'il y ait un MOM disponible (MQSeries par exemple) pour que cette application puisse envoyer et recevoir des messages. Avec un système de bridge (pont), on peut alors relier les files d'attente de ces deux providers (MQSeries↔GlassFish) et laisser les deux applications s'échanger des messages de manière transparente.

- ▶ <http://www.activemq.org/site/jms-to-jms-bridge.html>
- ▶ http://edocs.bea.com/wls/docs81/ConsoleHelp/messaging_bridge.html

GLASSFISH MDB stockés dans un pool

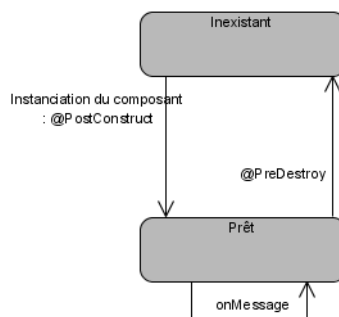
Bien que les spécifications n'obligent pas les conteneurs à avoir un pool de message-driven beans, la plupart des serveurs d'applications en utilise un pour augmenter les performances. C'est le cas de GlassFish qui les stocke dans un pool configurable. Sa taille est paramétrable ainsi que sa taille minimale et maximale. Cette configuration est faite via la console d'administration (menus *Configuration>EJB Container>MDB Settings*).

Figure 10-4

Cycle de vie d'un message-driven bean

Le cycle de vie d'un MDB

Le cycle de vie d'un message-driven bean est identique à celui d'un stateless bean. Il n'a que deux états : il existe ou il n'existe pas. Lorsqu'il existe, il est à l'écoute d'une destination JMS, prêt à traiter un message.



L'état inexistant signifie que le MDB n'a pas encore été instancié et qu'il n'existe pas en mémoire. Le passage à l'état prêt se fait lorsque le conteneur intercepte l'arrivée d'un message et invoque la méthode `onMessage` du MDB.

Les annotations de callback

Grâce aux annotations de callback, le conteneur d'EJB laisse la possibilité aux développeurs d'effectuer des traitements lors du passage d'un état à un autre. Il existe deux annotations utilisables par les message-driven beans :

- `@javax.annotation.PostConstruct`
- `@javax.annotation.PreDestroy`

Après avoir instancié un message-driven bean, le conteneur exécute les méthodes annotées par `@PostConstruct`. Dans le cas où le conteneur supprime l'EJB de la mémoire, les méthodes annotées `@PreDestroy` sont appelées.

JavaMail

Lorsque le bon de commande est créé, le système envoie un e-mail récapitulatif au client. JavaMail est l'API qui nous permet d'utiliser le courrier électronique.

Le courrier électronique repose sur le concept de clients et de serveurs. Les clients de mail (tels que Outlook, Messenger, Eudora, etc.) s'appuient sur un serveur de messagerie pour obtenir et envoyer des e-mails. Ces échanges sont normalisés par des protocoles particuliers (SMTP, POP3, etc.).

Les principaux protocoles de messagerie

- **SMTP** (*Simple Mail Transport Protocol*), protocole défini par la recommandation RFC 821, permet l'envoi d'e-mails vers un serveur.
- **POP3** (*Post Office Protocol*), défini par la recommandation RFC 1939, permet la réception d'e-mails. Protocole très populaire sur Internet, il définit une boîte aux lettres unique pour chaque utilisateur.
- **IMAP** (*Internet Message Access Protocol*), défini par la recommandation RFC 2060, permet la réception d'e-mails. Ce protocole est plus complexe car il apporte des fonctionnalités supplémentaires : plusieurs répertoires par utilisateur, partage de répertoires entre plusieurs utilisateurs, maintien des messages sur le serveur, etc.
- **NNTP** (*Network News Transport Protocol*) est utilisé par les forums de discussion (news).

Les RFC (*Request for Comments*) se trouvent à l'adresse suivante :

► <http://www.ietf.org/rfc.html>

Approfondir JavaMail

- <http://java.sun.com/products/javamail/>
- <http://java.sun.com/developer/onlineTraining/JavaMail/contents.html>
- <http://www.javaworld.com/jw-10-2001/jw-1026-javamail.html>

L'API JavaMail permet de s'abstraire de tout système de mail et d'utiliser la plupart des protocoles de communication de manière transparente. Ce n'est pas un serveur d'e-mails, mais un outil pour interagir avec

JAVAMAIL Un serveur SMTP

Pour pouvoir envoyer un e-mail, il vous faut connaître les paramètres de connexion d'un serveur de messagerie. Ce livre utilise le serveur SMTP du fournisseur d'accès Free, mais vous pouvez utiliser le serveur de votre choix, il suffit de changer les paramètres.

JAVAMAIL Le type MIME

Le type MIME (*Multipurpose Internet Mail Extensions*) est un standard permettant d'étendre les possibilités du courrier électronique, comme la possibilité d'insérer des documents (images, sons, texte, etc.) dans un courrier.

JAVAMAIL Les destinataires

Lorsqu'on envoie un e-mail, l'adresse du destinataire peut être typée :

- `RecipientType.TO` : destinataire direct ;
- `RecipientType.CC` : copie conforme ;
- `RecipientType.BCC` : copie cachée.

le serveur de messagerie. Les applications développées avec JavaMail sont ainsi comparables aux différentes messageries rencontrées, telles que Outlook, Lotus, Eudora, etc. Cette API propose donc des méthodes pour lire ou envoyer des e-mails, rechercher un message, etc. Les classes et interfaces de cette API sont regroupées dans le paquetage `javax.mail`.

Dans notre application, nous utiliserons le cas simple d'un envoi d'e-mail par SMTP. Pour cet exemple, nous n'utiliserons pas toute la panoplie des classes et interfaces de l'API, mais juste les principales, c'est-à-dire : `Session`, `Message`, `Transport` et `InternetAddress`.

La classe Session

À la manière de JMS, JavaMail possède une classe `javax.mail.Session` qui établit la connexion avec le serveur de messagerie. C'est elle qui encapsule les données liées à la connexion (options de configuration, login, mot de passe, nom du serveur) et à partir de laquelle les actions sont réalisées.

Création d'une session JavaMail

```
Properties properties = new Properties();
properties.put("mail.smtp.host", "smtp.free.fr");
properties.put("mail.smtp.auth", "true");
Session session = Session.getInstance(properties, null); ❶
```

Pour créer une session, on utilise la méthode `getInstance` ❶ à laquelle on passe les paramètres d'initialisation.

La classe Message

La classe `javax.mail.Message` est une classe abstraite qui encapsule le contenu du courrier électronique. Un message est composé d'un en-tête qui contient l'adresse de l'auteur et du destinataire, le sujet, etc. et d'un corps qui contient les données à envoyer. JavaMail fournit en standard une classe fille nommée `javax.mail.internet.MimeMessage` pour les messages possédant un type MIME.

La classe `Message` possède de nombreuses méthodes pour initialiser les données du message. Nous utiliserons les principales qui permettent de positionner l'adresse de l'émetteur ❶, du destinataire ❷, le sujet ❸ et le corps du message ❹ ainsi que la date d'envoi ❺.

Création d'un message

```
Message msg = new MimeMessage(session);
msg.setFrom(new InternetAddress("adresse@emetteur.com")); ❶
msg.setRecipients(Message.RecipientType.TO, ❷
    new InternetAddress("adresse@destinataire.com"));
msg.setSubject("Confirmation de commande"); ❸
msg.setText("La commande n°1002 a été créée"); ❹
msg.setSentDate(new Date()); ❺
```

La classe InternetAddress

La classe `javax.mail.internet.InternetAddress` est nécessaire pour chaque émetteur et destinataire d'e-mail. Elle hérite de la classe `javax.mail.Address` et représente une adresse e-mail au format `contact@serveurmail.com`. Pour créer une adresse e-mail, il suffit de passer une chaîne de caractères au constructeur.

Utilisation des adresses

```
Message msg = new MimeMessage(session);
msg.setFrom(new InternetAddress("adresse@emetteur.com"));
msg.setRecipients(Message.RecipientType.TO,
    new InternetAddress("adresse@destinataire.com"));
```

La classe Transport

La classe `javax.mail.Transport` se charge d'envoyer le message avec le protocole adéquat. Dans notre cas, pour SMTP, il faut obtenir un objet `Transport` dédié à ce protocole en utilisant la méthode `getTransport("smtp")` d'un objet `Session` ❶. Il faut ensuite établir la connexion en utilisant la méthode `connect()` ❷ en passant le nom du serveur de messagerie, le nom de l'utilisateur et son mot de passe. Pour envoyer le message que l'on aura créé antérieurement, il faut utiliser la méthode `sendMessage()` ❸ en lui passant la liste des destinataires `getAllRecipients()`. Enfin, il faut fermer la connexion à l'aide de la méthode `close()` ❹.

Envoi d'un e-mail

```
Transport transport = session.getTransport("smtp"); ❶
transport.connect("smtp.free.fr", "user", "password"); ❷
transport.sendMessage(msg, msg.getAllRecipients()); ❸
transport.close(); ❹
```

ARCHITECTURE Attention au pare-feu

Si vous avez un firewall (pare-feu) sur votre machine, vérifiez bien qu'il autorise le protocole SMTP sur le port 25. Sinon, les e-mails seront bloqués et ne pourront pas être envoyés.

Les traitements asynchrones de YAPS Pet Store

L'application YAPS Pet Store comprend des traitements qui peuvent être longs : elle doit donc avoir recours au mode asynchrone pour ne pas pénaliser ses temps de réponse. Lorsqu'un client achète des animaux et qu'il confirme cet achat en ligne, un e-mail récapitulatif ses achats doit lui être envoyé, et le bon de commande doit être imprimé pour être archivé par la société. Ces deux traitements peuvent être extrêmement longs si l'imprimante est éteinte ou le serveur de messagerie hors service, par exemple. Ils seront donc traités de manière asynchrone par des message-driven beans.

Pour des raisons administratives, les employés doivent être alertés des commandes contenant des reptiles. Leur application Swing peut être arrêtée, ou les employés ne pas être connectés. Ils doivent donc pouvoir recevoir ces alertes une fois à leur poste de travail. Nous utiliserons un listener JMS sur l'application Swing qui sélectionnera les messages contenant une propriété particulière l'informant que la commande contient des reptiles.

Tous ces composants seront à l'écoute du même Topic. Lors de la création d'un bon de commande, le stateless session bean enverra un message contenant le bon de commande dans le Topic `jms/topic/order`. À l'autre bout, deux MDB s'occuperont d'imprimer le bon de commande et d'envoyer un e-mail, tandis que le client Swing affichera une alerte pour les commandes comportant des reptiles (figure 10-5).

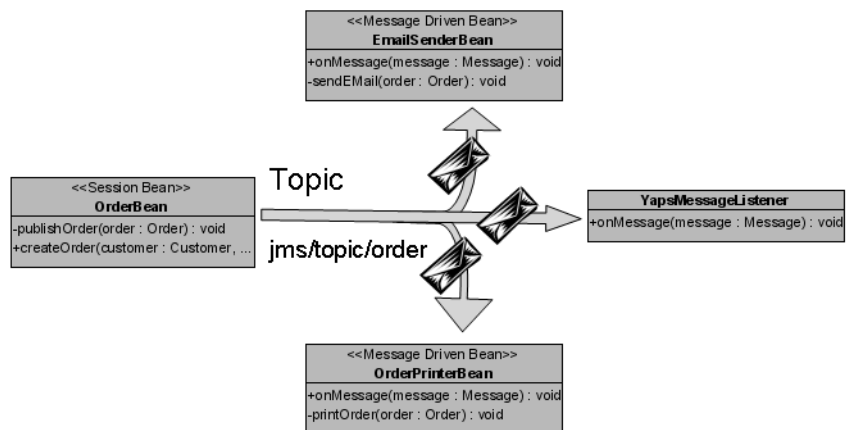


Figure 10-5
Le stateless bean envoie
un message dans le Topic.

L'envoi du message

L'envoi du message JMS se fait après la création du bon de commande. C'est donc l'EJB stateless `OrderBean` qui utilise l'API JMS pour envoyer le message contenant le bon de commande.

L'EJB `OrderBean` envoie un message

```
public class OrderBean implements OrderRemote, OrderLocal {

    @Resource(mappedName = "jms/petstoreConnectionFactory")
    private ConnectionFactory connectionFactory;
    @Resource(mappedName = "jms/topic/order")
    private Topic destinationOrder;

    @PostConstruct
    public void openConnection() {
        try {
            connection = connectionFactory.createConnection();
        } catch (JMSEException e) {
            throw new EJBException(e);
        }
    }

    @PreDestroy
    public void closeConnection() {
        if (connection != null) {
            try {
                connection.close();
            } catch (JMSEException e) {
                throw new EJBException(e);
            }
        }
    }

    public Order createOrder(Customer customer,
        Address deliveryAddress, CreditCard creditCard,
        List<CartItem> cartItems) {

        Order order = new Order(customer,
            em.merge(deliveryAddress), creditCard);
        List<OrderLine> orderLines = new ArrayList<OrderLine>();

        for (CartItem cartItem : cartItems) {
            orderLines.add(new OrderLine(cartItem.getQuantity(),
                cartItem.getItem()));
        }
        order.setOrderLines(orderLines);
        em.persist(order);
        publishOrder(order);
        return order;
    }
}
```

ARCHITECTURE Le couplage lâche

La logique métier pour créer un bon de commande a considérablement changé entre sa première implémentation et maintenant. La couche de stateless bean a été régulièrement mise à jour, mais pas les clients. C'est un des avantages d'un couplage lâche entre les couches : une partie du système peut être mise à jour sans avoir d'effet sur une autre.

◀ Injection de la fabrique de connexions et du Topic déclarés dans `GlassFish`.

◀ Grâce aux annotations de callback, on crée une connexion au provider JMS, à l'instanciation du stateless bean.

◀ Lorsque le stateless bean est supprimé de la mémoire, on libère les ressources JMS en clôturant la connexion.

◀ À l'aide de l'entity manager, on persiste l'entity bon de commande ainsi que ses lignes de commande.

◀ On envoie un message dans le Topic.

- On crée une session JMS avec un acquittement automatique des messages.
- On crée un MessageProducer qui, dans notre cas, sera un TopicPublisher.
- On crée un message de type objet.
- Dans les propriétés du message, on rajoute les catégories que l'on trouve dans le bon de commande (chiens, chats, reptiles, etc.).
- On affecte au corps du message l'entity bon de commande.
- Le message est envoyé dans le Topic.
- On referme la session et la connexion.

```
private void publishOrder(Order order) {
    Session session = null;
    try {
        session = connection.createSession(true,
            Session.AUTO_ACKNOWLEDGE);

        MessageProducer producer =
            session.createProducer(destinationOrder);

        ObjectMessage objectMessage =
            session.createObjectMessage();

        Set<Category> categories = order.getDistinctCategories();
        for (Category c : categories) {
            objectMessage.setBooleanProperty(c.getName(), true);
        }

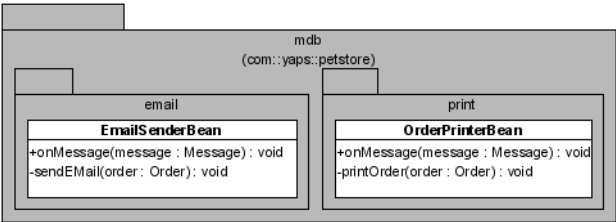
        objectMessage.setObject(order);

        producer.send(objectMessage);
    } catch (JMSException e) {
        throw new EJBException(e);
    } finally {
        session.close();
        // gestion des exceptions
    }
}
```

Les message-driven beans

Une fois le message JMS envoyé, les message-driven beans, qui sont à l'écoute du Topic, peuvent traiter le contenu du message.

Figure 10–6
Diagramme de classes
des message-driven beans



Envoi d'e-mails

Le MDB EmailSenderBean a pour rôle d'envoyer un e-mail à chaque réception de messages. Cet e-mail, à destination du client, récapitule le contenu de sa commande. Voici un exemple :

Subject : [YAPS] Confirmation: Order #1002 ①
Dear Paul Smith, ②

your order #1002 has been successfully placed.

Your shopping cart content is:

Goldfish female * 2 ❸

Iguana male *5

Looking forward to serve you again,

The YAPS team.

Pour constituer cet e-mail, le MDB a besoin de connaître le numéro du bon de commande ❶, le nom du client ❷ ainsi que son adresse e-mail ❹ et le contenu de son panier électronique ❸.

Extrait du MDB EmailSenderBean

```
@MessageDriven(mappedName = "jms/topic/order")
public class EmailSenderBean implements MessageListener {

    private static final String SMTP_HOST = "smtp.free.fr";
    private static final String USER = "yaps.petstore";
    private static final String PASSWORD = "yapspwd";

    public void onMessage(Message message) {
        try {
            if (message instanceof ObjectMessage) {
                ObjectMessage msg = (ObjectMessage) message;
                Order order = (Order) msg.getObject();
                sendEMail(order);
            }
        } catch (JMSEException e) {
        } catch (MessagingException e) {
            // gestion des exceptions
        }
    }

    private void sendEMail(Order order) throws MessagingException{

        Properties properties = new Properties();
        properties.put("mail.smtp.host", SMTP_HOST);
        properties.put("mail.smtp.auth", "true");
        Session session = Session.getInstance(properties, null);
        Message msg = new MimeMessage(session);
        msg.setFrom(new InternetAddress("no-reply@petstore.org"));
        String email = order.getCustomer().getEmail(); ❹
        msg.setRecipients(Message.RecipientType.TO,
            InternetAddress.parse(email, false));
        msg.setSentDate(new Date());
        msg.setSubject("[YAPS] Confirmation: Order #" + ❶
            order.getId());
```

◀ MDB à l'écoute du Topic jms/topic/order.

◀ Propriétés pour accéder au serveur de messagerie.

◀ La méthode onMessage est le point d'entrée du message-driven bean. Celui-ci reçoit un message du Topic et en récupère le contenu qui se trouve être un entity Order.

◀ Appelle une méthode privée pour envoyer le message.

◀ On crée une session JavaMail.

◀ Le message JavaMail est envoyé à l'adresse e-mail du client.

◀ Le sujet de l'e-mail.

Une méthode privée crée le corps du message.

L'e-mail est envoyé au serveur de messagerie par protocole SMTP.

```
msg.setText(formatBody(order));
```

```
Transport transport = session.getTransport("smtp");
transport.connect(SMTP_HOST, USER, PASSWORD);
transport.sendMessage(msg, msg.getAllRecipients());
// gestion des exceptions
transport.close();
```

```
}
```

Impression du bon de commande

Un autre MDB est à l'écoute du Topic : l'`OrderPrinterBean`. À chaque message reçu, ce message-driven bean imprime un bon de commande. L'impression en Java est fastidieuse et ce n'est pas le thème du livre (voir encadré ci-après). Dans l'exemple de code suivant, l'API de logging (journalisation) est utilisée pour afficher les données du bon de commande. Vous pourrez consulter les logs du serveur GlassFish pour en voir le descriptif.

Extrait du MDB `OrderPrinterBean`

MDB à l'écoute du Topic `jms/topic/order`.

Le logger fait office d'impression.

La méthode `onMessage` reçoit un message du Topic et en récupère le contenu qui est un bon de commande.

Appelle une méthode privée pour imprimer le bon de commande.

L'impression du bon de commande utilise l'API de logging Java pour afficher les informations.

```
@MessageDriven(mappedName = "jms/topic/order")
public class OrderPrinterBean implements MessageListener {

    private Logger logger =
        Logger.getLogger("com.yaps.petstore.mdb");

    public void onMessage(Message message) {
        try {
            if (message instanceof ObjectMessage) {
                ObjectMessage msg = (ObjectMessage) message;
                Order order = (Order) msg.getObject();

                printOrder(order);
            }
        } catch (JMSEException e) {
            throw new EJBException(e);
        }
    }

    private void printOrder(Order order) {
        logger.info("Order # " + order.getId() + " on the " +
            dateFormat.format(order.getOrderDate()));
        logger.info(order.getCustomer().getFirstname() +
            order.getCustomer().getLastname() + " bought ");
        for (OrderLine line : order.getOrderLines()) {
            logger.info("\t" + line.getItem().getName() + "*" +
                line.getQuantity() + "=" + line.getSubTotal());
        }
        logger.info("Total=" + order.getTotal());
    }
}
```

RETOUR D'EXPÉRIENCE L'impression en Java

L'impression en Java natif est un exercice complexe et fastidieux. Il suffit, pour s'en rendre compte, de consulter les classes et interfaces des sous-paquetages `javax.print`. Pour pallier ce problème, une multitude d'outils (Open Source ou non) vient aider le développeur dans cette lourde tâche. Le problème de ces outils est qu'ils ont tous leur propre manière de faire. Il est alors impossible de les interchanger si besoin.

Une autre possibilité, beaucoup plus portable, est d'utiliser XML et les transformations XSL (*eXtensible Stylesheet Language*). Nous avons vu dans le précédent chapitre qu'en utilisant les annotations JAXB, il était facile de transformer une grappe d'objets en flux XML. C'est ce que nous aurions pu faire avec l'entity Order. En plus des annotations JPA pour le rendre persistant, nous aurions pu rajouter des annotations JAXB pour obtenir une représentation XML du bon de commande. Ensuite, en effectuant une transformation XSL, on aurait pu transformer ce flux en page web ou en document PDF. Une fois le fichier PDF obtenu, on peut alors utiliser l'API `javax.print` pour l'envoyer vers une imprimante. L'utilitaire Open Source d'Apache FOP (*Formatting Objects Processor*) est souvent utilisé dans les projets pour simplifier les transformations XSL. Il permet très facilement de produire un document PDF, SVG, TXT, etc. à partir d'un document XML.

FOP :

- ▶ <http://xmlgraphics.apache.org/fop/>
- ▶ <http://www.onjava.com/pub/a/onjava/2002/10/16/fop.html>

D'autres outils d'impression :

ReportCat

- ▶ <http://www.netcat.li/java-report-printing-library/>

Java Print Dialog Framework

- ▶ <http://www.softframeworks.com>

RReport

- ▶ http://www.java4less.com/print_java_e.htm

Crystal Reports

- ▶ <http://www.businessobjects.com/product/catalog/crystalreports/>

Listener JMS de l'application Swing

L'interface homme-machine doit, elle aussi, être à l'écoute du Topic. En fait, elle affiche en temps réel uniquement les informations des bons de commande contenant des reptiles, et ceci grâce au message selector de JMS.

Un nouveau sous-menu *Watch orders* affiche un composant JTable qui se rafraîchit automatiquement à l'arrivée d'un message. Pour ne pas bloquer l'affichage, cette fenêtre s'exécute dans un Thread à part.

Comme nous l'avons déjà expliqué, pour des raisons pédagogiques, l'application graphique n'utilise pas le conteneur client (ACC). Pour récupérer la référence vers la fabrique de connexions JMS et le Topic, elle doit donc utiliser JNDI pour les localiser. Pour masquer ces appels à l'API JNDI, nous pouvons enrichir le Service Locator que nous avons vu au chapitre 6, *Exécution de l'application*.

Design pattern ServiceLocator.	▶
Système de cache du service locator.	▶
Retourne une fabrique de connexions JMS.	▶
Retourne une destination JMS.	▶
Méthode privée permettant de retrouver un objet dans le cache ou dans JNDI.	▶

Extrait du ServiceLocator avec les nouvelles méthodes

```
public class ServiceLocator {  
    private Context initialContext;  
    private Map<String, Object> cache;  
  
    public ConnectionFactory getConnectionFactory(  
        String connFactoryName) throws ServiceLocatorException {  
        ConnectionFactory factory = (ConnectionFactory)  
            getRemoteObject(connFactoryName);  
        return factory;  
    }  
  
    public Destination getDestination(String destinationName) {  
        Destination destination = (Destination)  
            getRemoteObject(destinationName);  
        return destination;  
    }  
  
    private synchronized Object getRemoteObject(String jndiName)  
        throws ServiceLocatorException {  
        Object remoteObject = cache.get(jndiName);  
        if (remoteObject == null) {  
            try {  
                remoteObject = initialContext.lookup(jndiName);  
                cache.put(jndiName, remoteObject);  
            } catch (Exception e) {  
                throw new ServiceLocatorException(e);  
            }  
        }  
        return remoteObject;  
    }  
}
```

L'application Swing n'a plus qu'à utiliser le ServiceLocator pour obtenir la fabrique de connexions et la destination sur laquelle écouter.

Reception d'un message

On déclare la fabrique de connexions et le Topic.	▶
Utilisation du service locator pour obtenir la fabrique de connexions JMS et le Topic.	▶
On se connecte au provider de messages.	▶
On crée une session.	▶

```
public class YapsMsgListener implements MessageListener {  
    private ConnectionFactory connectionFactory;  
    private Topic destinationOrder;  
  
    private void receiveOrder(Order order) {  
        // Pour simplifier la lecture du code, la réception  
        // asynchrone à l'aide d'un thread n'est pas implémentée  
        connectionFactory = ServiceLocator.getInstance().  
            getConnectionFactory("jms/petstoreConnectionFactory");  
        destinationOrder = ServiceLocator.getInstance().  
            getDestination("jms/topic/order");  
  
        Connection connection =  
            connectionFactory.createConnection();  
  
        Session session = connection.createSession(true,  
            Session.AUTO_ACKNOWLEDGE);
```

```

MessageConsumer consumer = session.createConsumer
    (destinationOrder,"Reptiles=true");

consumer.setMessageListener(this);

connection.start();

}

public void onMessage(Message message) {
    if (message instanceof ObjectMessage) {
        ObjectMessage objMsg = (ObjectMessage) message;
        tableModel.add(objMsg.getObject());
    }
}
}

```

◀ On crée un MessageProducer qui filtre les messages sur la propriété Reptiles=true.

◀ On associe un listener à la classe courante.

◀ On démarre la connexion.

◀ Les messages arrivent par la méthode onMessage et sont affichés dans une JTable.

Le résultat graphique est le suivant (figure 10–7). Les commandes contenant des reptiles s'affichent au fur et à mesure de leur arrivée dans une liste. En sélectionnant une commande et en cliquant sur le bouton *View* on peut en connaître le détail.

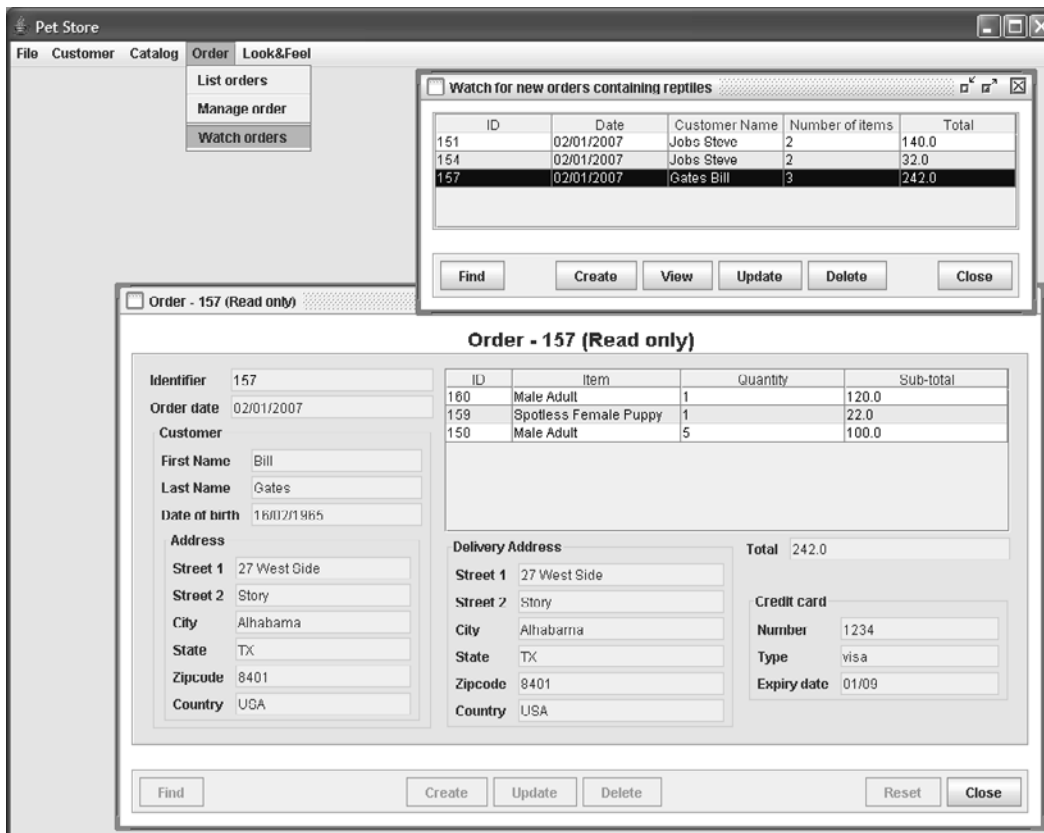


Figure 10–7 Application Swing recevant les messages

Paquetages des MDB

Architecture

The diagram illustrates the architecture of a Pet Store System, organized into several packages: **ui**, **stateless**, **entity**, **stateful**, **mdb**, **petex**, and **barkbank**.

- ui Package:** Contains three boundary components: `PetstoreFrame` (which depends on `JMS`), `CatalogDelegate`, and `CustomerDelegate`. It also includes a utility component `ServiceLocator` that provides a `lookup JNDI` service.
- stateless Package:** Contains three component interfaces: `CatalogBean`, `CustomerBean`, and `OrderBean`. `CatalogBean` and `CustomerBean` are provided by `Remote` endpoints, while `OrderBean` is provided by `Local` endpoints.
- entity Package:** Contains several entity components: `Category`, `Product`, `Item`, `Customer`, `Address`, `OrderLine`, `Order`, and `CreditCard`. These are interconnected with multiplicity constraints (e.g., 1 to 0..*, 1 to 1, 1 to 1..*).
- stateful Package:** Contains a component interface `ShoppingCartBean` provided by a `Local` endpoint, which depends on the `entity` package.
- mdb Package:** Contains two component interfaces: `EmailSenderBean` (provided by `JMS`) and `OrderPrinterBean` (provided by `JMS`).
- petex Package:** Contains a component interface `Delivery` provided by an `HTTP` endpoint.
- barkbank Package:** Contains a component interface `Validation` provided by an `HTTP` endpoint.
- Control Layer:** A central package containing `AccountController`, `CatalogController`, and `ShoppingCartController`. These controllers interact with the `ui` package (via `Remote` and `Local` endpoints), the `stateless` package (via `Remote` and `Local` endpoints), the `stateful` package (via `Local` endpoint), and the `mdb`, `petex`, and `barkbank` packages (via `JMS` and `HTTP` endpoints).

A small screenshot of the Pet Store application's graphical user interface is shown in the bottom left corner.

292

Exécuter l'application

Pour exécuter l'application, il faut utiliser les mêmes tâches Ant `yaps-clean`, `yaps-build` et `yaps-deploy` qui compileront, packageront les classes et déploieront les fichiers archives. Les message-driven beans sont déployés dans un fichier `.jar` à part : `mdb.jar` qui est inclus dans le `petstore.ear`.

Vous pouvez maintenant utiliser l'application web pour acheter des animaux domestiques. Ceci aura pour conséquence d'envoyer un e-mail en utilisant votre serveur de messagerie et d'afficher les informations du bon de commande dans les logs GlassFish. Si vous achetez un reptile, vous verrez l'interface Swing recevoir un événement.

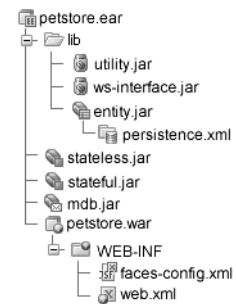


Figure 10–10 Contenu du fichier `petstore.ear`

En résumé

Ce chapitre complète le développement de l'application YAPS Pet Store. Lorsque les clients valident leurs achats sur le site, le système doit envoyer un e-mail de confirmation, imprimer un bon de commande et alerter les employés des achats contenant des reptiles. Tous ces traitements se font de manière asynchrone en utilisant JMS et les message-driven beans.

Spécifications Java EE 5



Les spécifications Java EE s'accumulent ou se subdivisent. Certaines disparaissent tandis que d'autres naissent. C'est pourquoi chacune d'entre elles possède un numéro de version et un numéro de JSR (*Java Specification Request*). Pour s'y retrouver plus aisément, les tableaux ci-dessous dressent une liste exhaustive des spécifications qui composent à ce jour Java EE 5.

Tableau A-1 Spécifications Java Enterprise Edition 5

Spécification	Version	JSR	URL
Java EE (Java Platform, Enterprise Edition)	5.0	244	http://jcp.org/en/jsr/detail?id=244

Tableau A-2 Spécifications Web Services

Spécification	Version	JSR	URL
JAX-RPC (Java APIs for XML-based RPC)	1.1	101	http://jcp.org/en/jsr/detail?id=101
JAX-WS (Java API for XML-based Web Services)	2.0	224	http://jcp.org/en/jsr/detail?id=224
JAXB (Java Architecture for XML Binding)	2.0	222	http://jcp.org/en/jsr/detail?id=222
SAAJ (Soap with Attachments API for Java)	1.0	67	http://jcp.org/en/jsr/detail?id=67
StAX (Streaming API for XML)	1.0	173	http://jcp.org/en/jsr/detail?id=173
Web Services	1.2	109	http://jcp.org/en/jsr/detail?id=109
Web Services Metadata	2.0	181	http://jcp.org/en/jsr/detail?id=181

Tableau A-3 Spécifications web

Spécification	Version	JSR	URL
JSF (JavaServer Faces)	1.2	252	http://jcp.org/en/jsr/detail?id=252
JSP (JavaServer Pages)	2.1	245	http://www.jcp.org/en/jsr/detail?id=245
JSTL (JavaServer Pages Standard Tag Library)	1.2	52	http://jcp.org/en/jsr/detail?id=52

Tableau A-3 Spécifications web

Spécification	Version	JSR	URL
Servlet	2.5	154	http://jcp.org/en/jsr/detail?id=154

Tableau A-4 Spécifications Enterprise

Spécification	Version	JSR	URL
Common Annotations	1.0	250	http://jcp.org/en/jsr/detail?id=250
EJB (Enterprise JavaBeans)	3.0	220	http://jcp.org/en/jsr/detail?id=220
JAF (JavaBeans Activation Framework)	1.1	925	http://jcp.org/en/jsr/detail?id=925
JavaMail	1.4	919	http://jcp.org/en/jsr/detail?id=919
JCA (J2EE Connector Architecture)	1.5	112	http://jcp.org/en/jsr/detail?id=112
JMS (Java Message Service)	1.1	914	http://jcp.org/en/jsr/detail?id=914
JPA (Java Persistence API)	1.0	220	http://jcp.org/en/jsr/detail?id=220
JTA (Java Transaction API)	1.1	907	http://jcp.org/en/jsr/detail?id=907

Tableau A-5 Spécifications Management et sécurité

Spécification	Version	JSR	URL
JACC (Java Authorization Contract for Containers)	1.1	115	http://jcp.org/en/jsr/detail?id=115
Java EE Application Deployment	1.2	88	http://jcp.org/en/jsr/detail?id=88
Java EE Management	1.1	77	http://jcp.org/en/jsr/detail?id=77

Tâches Ant

TÉLÉCHARGER **Ant 1.7**

► <http://ant.apache.org/>

Ce livre utilise des tâches Ant pour compiler, packager, déployer les applications (fichier `build.xml`) ainsi que pour administrer le serveur GlassFish (fichier `admin.xml`). Vous retrouverez ces fichiers sur le site www.antoniogoncalves.org. Pour pouvoir exécuter ces tâches, vous devez avoir installé Ant au préalable et ajouté le répertoire `%ANT_HOME%/bin` dans votre variable `PATH`.

Build.xml

Dans ce fichier, vous trouverez toutes les tâches utilisées pour les applications YAPS Pet Store, BarkBank et PetEx. Pour les exécuter, il suffit de taper la ligne de commande suivante :

```
ant <le nom de la tâche>
```

Vous trouverez dans le fichier ci-dessous la liste des tâches que vous pouvez exécuter.

Variables globales de l'application.

Variables propres de l'application YAPS Pet Store.

```
<?xml version="1.0"?>
<project name="Petstore" default="core">
  <property name="application.name" value="petstore"/>
  <property name="home.dir" value="."/>
  <property name="barkbank.home" value="${home.dir}/barkbank"/>
  <property name="petex.home" value="${home.dir}/petex"/>
  <property name="yaps.home" value="${home.dir}/yaps"/>
  <property name="yaps.build.dir" value="${yaps.home}/build"/>
  <property name="yaps.config.dir" value="${yaps.home}/config"/>
  <property name="yaps.src.dir" value="${yaps.home}/src"/>
  <property name="yaps.web.dir" value="${yaps.home}/resources"/>
  <property name="yaps.web-inf.dir"
    value="${yaps.home}/web-inf"/>
```


Variables propres de l'application BarkBank.

```

<property name="yaps.generated.src.dir"
  value="${yaps.home}/generated"/>
<property name="yaps.classes.dir"
  value="${yaps.home}/classes/production"/>
<property name="barkbank.build.dir"
  value="${barkbank.home}/build"/>
<property name="barkbank.src.dir"
  value="${barkbank.home}/src"/>
<property name="barkbank.web.dir"
  value="${barkbank.home}/resources"/>
<property name="barkbank.web-inf.dir"
  value="${barkbank.home}/web-inf"/>
<property name="barkbank.generated.src.dir"
  value="${barkbank.home}/generated"/>
<property name="barkbank.classes.dir"
  value="${barkbank.home}/classes/production"/>

```

Variables propres de l'application PetEx.

```

<property name="petex.build.dir" value="${petex.home}/build"/>
<property name="petex.src.dir" value="${petex.home}/src"/>
<property name="petex.web.dir"
  value="${petex.home}/resources"/>
<property name="petex.web-inf.dir"
  value="${petex.home}/web-inf"/>
<property name="petex.generated.src.dir"
  value="${petex.home}/generated"/>
<property name="petex.classes.dir"
  value="${petex.home}/classes/production"/>

```

Fichiers pour le déploiement de l'application YAPS Pet Store.

```

<property name="yaps.client.jar"
  value="${yaps.build.dir}/${application.name}.jar"/>
<property name="yaps.utility.jar"
  value="${yaps.build.dir}/lib/utility.jar"/>
<property name="yaps.ws.jar"
  value="${yaps.build.dir}/lib/ws-interface.jar"/>
<property name="yaps.entity.jar"
  value="${yaps.build.dir}/lib/entity.jar"/>
<property name="yaps.mdb.jar"
  value="${yaps.build.dir}/mdb.jar"/>
<property name="yaps.stateless.jar"
  value="${yaps.build.dir}/stateless.jar"/>
<property name="yaps.stateful.jar"
  value="${yaps.build.dir}/stateful.jar"/>
<property name="yaps.war"
  value="${yaps.build.dir}/${application.name}.war"/>
<property name="yaps.ear"
  value="${yaps.build.dir}/${application.name}.ear"/>

```

Fichier pour le déploiement de l'application BarkBank.

```

<property name="barkbank.war"
  value="${barkbank.build.dir}/barkbank.war"/>

```

Fichier pour le déploiement de l'application PetEx.

```

<property name="petex.war"
  value="${petex.build.dir}/petex.war"/>

```

```

<property environment="env"/>
<property name="glassfish.home"
  value="${env.GLASSFISH_HOME}"/>
<property name="glassfish.lib" value="${glassfish.home}/lib"/>
<property name="derby.home" value="${glassfish.home}/javadb"/>
<property name="derby.lib" value="${derby.home}/lib"/>
<property name="asadmin"
  value="${glassfish.home}/bin/asadmin.bat"/>
<property name="wsген"
  value="${glassfish.home}/bin/wsген.bat"/>
<property name="wsimport"
  value="${glassfish.home}/bin/wsimport.bat"/>
<property name="echo" value="false"/>
<property name="verifier"
  value="${glassfish.home}/bin/verifier.bat"/>
<property name="verifier.dir" value="${home.dir}/verifier"/>

```

◀ Variables d'environnement et utilitaires d'administration et de génération de GlassFish.

```

<property name="server.user.name" value="admin"/>
<property name="server.passwordfile" value="passwordfile"/>
<property name="server.host" value="localhost"/>
<property name="server.port" value="8080"/>
<property name="server.admin.port" value="8282"/>

```

◀ Propriétés du serveur GlassFish.

```

<property name="db.host" value="localhost"/>
<property name="db.port" value="1527"/>
<property name="db.sid" value="${application.name}DB"/>
<property name="db.user" value="dbuser"/>
<property name="db.password" value="dbpwd"/>
<property name="db.driver"
  value="org.apache.derby.jdbc.ClientDriver"/>
<property name="db.url"
  value="jdbc:derby://${db.host}:${db.port}/${db.sid}"/>

```

◀ Propriétés de la base de données Derby.

```

<path id="classpath">
  <pathelement location="${glassfish.lib}/
    ➤ appserv-deployment-client.jar"/>
  <pathelement location="${glassfish.lib}/appserv-rt.jar"/>
  <pathelement location="${glassfish.lib}/
    ➤ webservices-rt.jar"/>
  <pathelement location="${glassfish.lib}/appserv-admin.jar"/>
  <pathelement location="${glassfish.lib}/javaee.jar"/>
  <pathelement location="${glassfish.lib}/
    ➤ toplink-essentials-agent.jar"/>
  <pathelement location="${glassfish.lib}/
    ➤ toplink-essentials.jar"/>
  <pathelement location="${glassfish.lib}/install/
    ➤ applications/jmsra/imqjmsra.jar"/>
  <pathelement location="${derby.lib}/derbyclient.jar"/>
</path>

```

◀ Classes et JAR utilisés pour compiler et exécuter l'application.

```

<path id="yaps-classpath">
  <pathelement location="${yaps.classes.dir}"/>
</path>

```

◀ Classes de l'application YAPS Pet Store.

Classes de l'application BarkBank.

▶

```
<path id="barkbank-classpath">
  <pathelement location="${barkbank.classes.dir}"/>
</path>
```

Classes de l'application PetEx.

▶

```
<path id="petex-classpath">
  <pathelement location="${petex.classes.dir}"/>
</path>
```

Suppression des répertoires de travail des applications YAPS Pet Store, Barkbank et PetEx.

▶

```
<target name="clean">
  <antcall target="yaps-clean"/>
  <antcall target="barkbank-clean"/>
  <antcall target="petex-clean"/>
</target>
```

Suppression des répertoires de travail de l'application YAPS Pet Store.

▶

```
<target name="yaps-clean">
  <echo message="Cleans the Yaps environment"/>
  <delete dir="${yaps.home}/classes"/>
  <delete dir="${yaps.build.dir}"/>
</target>
```

Suppression des répertoires de travail de l'application Barkbank.

▶

```
<target name="barkbank-clean">
  <echo message="Cleans the BarkBank environment"/>
  <delete dir="${barkbank.home}/classes"/>
  <delete dir="${barkbank.build.dir}"/>
  <delete dir="${barkbank.generated.src.dir}"/>
</target>
```

Suppression des répertoires de travail de l'application PetEx.

▶

```
<target name="petex-clean">
  <echo message="Cleans the PetEx environment"/>
  <delete dir="${petex.home}/classes"/>
  <delete dir="${petex.build.dir}"/>
  <delete dir="${petex.generated.src.dir}"/>
</target>
```

Création des répertoires de travail des applications YAPS Pet Store, Barkbank et PetEx.

▶

```
<target name="prepare">
  <antcall target="yaps-prepare"/>
  <antcall target="barkbank-prepare"/>
  <antcall target="petex-prepare"/>
</target>
```

Création des répertoires de travail de l'application YAPS Pet Store.

▶

```
<target name="yaps-prepare">
  <echo message="Setup the Yaps environment"/>
  <mkdir dir="${yaps.classes.dir}"/>
  <mkdir dir="${yaps.build.dir}/lib"/>
</target>
```

Création des répertoires de travail de l'application BarkBank.

▶

```
<target name="barkbank-prepare">
  <echo message="Setup the Barkbank environment"/>
  <mkdir dir="${barkbank.classes.dir}"/>
  <mkdir dir="${barkbank.build.dir}"/>
  <mkdir dir="${barkbank.generated.src.dir}"/>
</target>
```

```
<target name="petex-prepare">
  <echo message="Setup the PetEx environment"/>
  <mkdir dir="${petex.classes.dir}"/>
  <mkdir dir="${petex.build.dir}"/>
  <mkdir dir="${petex.generated.src.dir}"/>
</target>
```

- ◀ Création des répertoires de travail de l'application PetEx.

```
<target name="compile" depends="prepare">
  <antcall target="yaps-compile"/>
  <antcall target="barkbank-compile"/>
  <antcall target="petex-compile"/>
</target>
```

- ◀ Compile le code source des applications YAPS Pet Store, Barkbank et PetEx.

```
<target name="yaps-compile" depends="yaps-prepare">
  <echo message="Compile the YAPS generated classes"/>
  <javac srcdir="${yaps.generated.src.dir}"
        destdir="${yaps.classes.dir}"
        deprecation="on">
<classpath refid="classpath"/>
<classpath refid="yaps-classpath"/>
  </javac>
  <echo message="Compile the YAPS classes"/>
  <javac srcdir="${yaps.src.dir}"
        destdir="${yaps.classes.dir}"
        deprecation="on">
<compilerarg value="-Xlint:unchecked"/>
<classpath refid="classpath"/>
<classpath refid="yaps-classpath"/>
  </javac>
</target>
```

- ◀ Compile le code source de l'application YAPS Pet Store.

```
<target name="barkbank-compile" depends="barkbank-prepare">
  <echo message="Compile the BarkBank classes"/>
  <javac srcdir="${barkbank.src.dir}"
        destdir="${barkbank.classes.dir}"
        deprecation="on">
<classpath refid="classpath"/>
<classpath refid="barkbank-classpath"/>
  </javac>
</target>
```

- ◀ Compile le code source de l'application Bark-Bank.

```
<target name="petex-compile" depends="petex-prepare">
  <echo message="Compile the Petex classes"/>
  <javac srcdir="${petex.src.dir}"
        destdir="${petex.classes.dir}"
        deprecation="on">
<classpath refid="classpath"/>
<classpath refid="petex-classpath"/>
  </javac>
</target>
```

- ◀ Compile le code source de l'application PetEx.

Compile le code source de l'application YAPS Pet Store.

```
<target name="yaps-compile" depends="yaps-prepare">
  <echo message="Compile the YAPS generated classes"/>
  <javac srcdir="${yaps.generated.src.dir}"
        destdir="${yaps.classes.dir}"
        deprecation="on">
    <classpath refid="classpath"/>
    <classpath refid="yaps-classpath"/>
  </javac>
  <echo message="Compile the YAPS classes"/>
  <javac srcdir="${yaps.src.dir}"
        destdir="${yaps.classes.dir}"
        deprecation="on">
    <compilerarg value="-Xlint:unchecked"/>
    <classpath refid="classpath"/>
    <classpath refid="yaps-classpath"/>
  </javac>
</target>
```

Package les applications YAPS Pet Store, Barkbank et PetEx.

```
<target name="build" depends="compile">
  <antcall target="yaps-build"/>
  <antcall target="barkbank-build"/>
  <antcall target="petex-build"/>
</target>
```

Package l'application YAPS Pet Store.

```
<target name="yaps-build"
  depends="yaps-compile,yaps-build-client-jar,yaps-build-
stateless-jar,yaps-build-stateful-jar,yaps-build-mdb-jar,yaps-
build-entity-jar,yaps-build-utility-jar,yaps-build-ws-
jar,yaps-build-war,yaps-build-ear"/>
```

Crée le JAR client.

```
<target name="yaps-build-client-jar">
  <echo message="Creates the Client jar"/>
  <jar jarfile="${yaps.client.jar}">
    <fileset dir="${yaps.classes.dir}">
      <include name="com/yaps/petstore/client/**/*.class"/>
      <include name="com/yaps/petstore/entity/**/*.class"/>
      <include name="com/yaps/petstore/util/**/*.class"/>
      <include name="com/yaps/petstore/exception/**/*.class"/>
      <include
        name="com/yaps/petstore/stateless/**/*.Remote.class"/>
      <include name="org/**/*.class"/>
    </fileset>
  </jar>
</target>
```

Crée le JAR contenant les stateless beans.

```
<target name="yaps-build-stateless-jar">
  <echo message="Creates the EJB stateless jar"/>
  <jar jarfile="${yaps.stateless.jar}">
    <fileset dir="${yaps.classes.dir}">
      <include name="com/yaps/petstore/stateless/**/*.class"/>
    </fileset>
  </jar>
</target>
```

```
<target name="yaps-build-stateful-jar">
  <echo message="Creates the EJB Stateful jar"/>
  <jar jarfile="${yaps.stateful.jar}">
    <fileset dir="${yaps.classes.dir}">
      <include name="com/yaps/petstore/stateful/**/*.class"/>
    </fileset>
  </jar>
</target>
```

◀ Crée le JAR contenant le stateful bean.

```
<target name="yaps-build-mdb-jar">
  <echo message="Creates the EJB MDB jar"/>
  <jar jarfile="${yaps.mdb.jar}">
    <fileset dir="${yaps.classes.dir}">
      <include name="com/yaps/petstore/mdb/**/*.class"/>
    </fileset>
  </jar>
</target>
```

◀ Crée le JAR contenant les message-driven beans.

```
<target name="yaps-build-entity-jar">
  <echo message="Creates the EJB Entity jar"/>
  <jar jarfile="${yaps.entity.jar}">
    <fileset dir="${yaps.classes.dir}">
      <include name="com/yaps/petstore/entity/**/*.class"/>
    </fileset>
    <fileset dir="${yaps.config.dir}">
      <include name="META-INF/persistence.xml"/>
    </fileset>
  </jar>
</target>
```

◀ Crée le JAR contenant les entités.

```
<target name="yaps-build-utility-jar">
  <echo message="Creates the Utility jar"/>
  <jar jarfile="${yaps.utility.jar}">
    <fileset dir="${yaps.classes.dir}">
      <include name="com/yaps/petstore/util/**/*.class"/>
      <include name="com/yaps/petstore/exception/**/*.class"/>
    </fileset>
  </jar>
</target>
```

◀ Crée le JAR contenant les classes utilitaires.

```
<target name="yaps-build-ws-jar">
  <echo message="Creates the WebService jar"/>
  <jar jarfile="${yaps.ws.jar}">
    <fileset dir="${yaps.classes.dir}">
      <include name="com/barkbank/**/*.class"/>
      <include name="com/petex/**/*.class"/>
    </fileset>
  </jar>
</target>
```

◀ Crée le JAR contenant les artefacts des services web.

Crée le WAR contenant l'application web.

```
> <target name="yaps-build-war">
    <echo message="Creates the PetStore Web Application"/>
    <war destfile="${yaps.war}"
        webxml="${yaps.web-inf.dir}/web.xml">
    <fileset dir="${yaps.web.dir}">
        <include name="**/*.jsp"/>
        <include name="**/*.jspx"/>
        <include name="**/*.gif"/>
        <include name="**/*.jpg"/>
        <include name="**/*.css"/>
    </fileset>
    <webinf dir="${yaps.web-inf.dir}">
        <include name="faces-config.xml"/>
    </webinf>
    <classes dir="${yaps.classes.dir}">
        <include name="com/yaps/petstore/jsf/**/*.class"/>
    </classes>
    </war>
</target>
```

Crée le EAR contenant toutes les librairies.

```
> <target name="yaps-build-ear">
    <echo message="Creates the PetStore Enterprise Application"/>
    <delete file="${yaps.ear}"/>
    <jar jarfile="${yaps.ear}" basedir="${yaps.build.dir}"
        excludes="${application.name}.jar"/>
</target>
```

Package l'application BarkBank.

```
> <target name="barkbank-build" depends="barkbank-
compile,barkbank-generate-server-artifacts">
    <echo message="Creates the Barkbank Web Application"/>
    <war destfile="${barkbank.war}"
        webxml="${barkbank.web-inf.dir}/web.xml">
    <fileset dir="${barkbank.web.dir}">
        <include name="**/*.jsp"/>
        <include name="**/*.gif"/>
    </fileset>
    <classes dir="${barkbank.classes.dir}">
        <include name="com/barkbank/**/*.class"/>
    </classes>
    </war>
</target>
```

Package l'application PetEx.

```
> <target name="petex-build" depends="petex-compile,
        petex-generate-server-artifacts">
    <echo message="Creates the Petex Web Application"/>
    <war destfile="${petex.war}"
        webxml="${petex.web-inf.dir}/web.xml">
    <fileset dir="${petex.web.dir}">
        <include name="**/*.jsp"/>
        <include name="**/*.gif"/>
    </fileset>
    <classes dir="${petex.classes.dir}">
        <include name="com/petex/**/*.class"/>
    </classes>
    </war>
</target>
```

```
<target name="generate-server-artifacts">
  <antcall target="barkbank-generate-server-artifacts"/>
  <antcall target="petex-generate-server-artifacts"/>
</target>
```

◀ Génère les artefacts serveur des services web de BarkBank et PetEx.

```
<target name="barkbank-generate-server-artifacts"
          depends="barkbank-compile">
  <echo message="Generates the BarkBank artifacts"/>
  <exec executable="${wsngen}" failonerror="true">
<arg line=" -cp ${barkbank.classes.dir}"/>
<arg line=" -d ${barkbank.classes.dir}"/>
<arg line=" -keep"/>
<arg line=" -wsdl"/>
<arg line=" -r ${barkbank.generated.src.dir}"/>
<arg line=" -s ${barkbank.generated.src.dir}"/>
<arg line=" com.barkbank.validator.Validation"/>
  </exec>
</target>
```

◀ Génère les artefacts serveur du service web de BarkBank.

```
<target name="petex-generate-server-artifacts"
          depends="petex-compile">
  <echo message="Generates the Petex artifacts"/>
  <exec executable="${wsngen}" failonerror="true">
<arg line=" -cp ${petex.classes.dir}"/>
<arg line=" -d ${petex.classes.dir}"/>
<arg line=" -keep"/>
<arg line=" -wsdl"/>
<arg line=" -r ${petex.generated.src.dir}"/>
<arg line=" -s ${petex.generated.src.dir}"/>
<arg line=" com.petex.transport.Delivery"/>
  </exec>
</target>
```

◀ Génère les artefacts serveur du service web de PetEx.

```
<target name="generate-client-artifacts">
  <antcall target="barkbank-generate-client-artifacts"/>
  <antcall target="petex-generate-client-artifacts"/>
</target>
```

◀ Génère les artefacts client des services web de BarkBank et PetEx.

```
<target name="barkbank-generate-client-artifacts"
          depends="yaps-prepare">
  <echo message="Generates client artifacts for Bark Bank"/>
  <exec executable="${wsimport}" failonerror="true">
<arg line=" -d ${yaps.classes.dir}"/>
<arg line=" -keep"/>
<arg line=" -s ${yaps.generated.src.dir}"/>
<arg line=" http://${server.host}:${server.port}/barkbank/
ValidationService?WSDL"/>
  </exec>
</target>
```

◀ Génère les artefacts client du service web de BarkBank.

Génère les artefacts client du service web de PetEx.

```
<target name="petex-generate-client-artifacts"
        depends="yaps-prepare">
    <echo message="Generates client artifacts for PetEx"/>
    <exec executable="${wsimport}" failonerror="true">
<arg line="-d ${yaps.classes.dir}"/>
<arg line="-keep"/>
<arg line="-s ${yaps.generated.src.dir}"/>
<arg line="http://${server.host}:${server.port}/petex/
DeliveryService?WSDL"/>
    </exec>
</target>
```

Déploiement des applications YAPS Pet Store, BarkBank et PetEx.

```
<target name="deploy">
    <antcall target="barkbank-build"/>
    <antcall target="barkbank-deploy"/>
    <antcall target="barkbank-generate-client-artifacts"/>
    <antcall target="petex-build"/>
    <antcall target="petex-deploy"/>
    <antcall target="petex-generate-client-artifacts"/>
    <antcall target="yaps-build"/>
    <antcall target="yaps-deploy"/>
</target>
```

Déploiement de l'application YAPS Pet Store et insertion des données en base.

```
<target name="yaps-deploy">
    <echo message="Deploys the YAPS application"/>
    <exec executable="${asadmin}">
<arg line="deploy"/>
<arg line="--echo=${echo}"/>
<arg line="--user ${server.user.name}"/>
<arg line="--passwordfile ${server.passwordfile}"/>
<arg line="--host ${server.host}"/>
<arg line="--port ${server.admin.port}"/>
<arg line="${yaps.ear}"/>
    </exec>

    <antcall target="db-insert-data"/>
</target>
```

Déploiement de l'application BarkBank.

```
<target name="barkbank-deploy">
    <echo message="Deploys the BarkBank application"/>
    <exec executable="${asadmin}">
<arg line="deploy"/>
<arg line="--echo=${echo}"/>
<arg line="--user ${server.user.name}"/>
<arg line="--passwordfile ${server.passwordfile}"/>
<arg line="--host ${server.host}"/>
<arg line="--port ${server.admin.port}"/>
<arg line="${barkbank.war}"/>
    </exec>
</target>
```

```

<target name="petex-deploy">
  <echo message="Deploys the PetEx application"/>
  <exec executable="${asadmin}">
    <arg line="deploy"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="${petex.war}"/>
  </exec>
</target>

<target name="undeploy">
  <antcall target="yaps-undeploy"/>
  <antcall target="barkbank-undeploy"/>
  <antcall target="petex-undeploy"/>
</target>

<target name="yaps-undeploy">
  <echo message="Undeploys the YAPS application"/>
  <exec executable="${asadmin}">
    <arg line="undeploy"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="${application.name}"/>
  </exec>
</target>

<target name="barkbank-undeploy">
  <echo message="Undeploys the BarkBank application"/>
  <exec executable="${asadmin}">
    <arg line="undeploy"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="barkbank"/>
  </exec>
</target>

<target name="petex-undeploy">
  <echo message="Undeploys the PetEx application"/>
  <exec executable="${asadmin}">
    <arg line="undeploy"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="petex"/>
  </exec>
</target>

```

◀ Déploiement de l'application PetEx.

◀ Suppression des applications YAPS Pet Store, BarkBank et PetEx.

◀ Suppression de l'application YAPS Pet Store.

◀ Suppression de l'application BarkBank.

◀ Suppression de l'application PetEx.

Vérifie les applications YAPS Pet Store, BarkBank et PetEx.	<pre> <target name="verify"> <antcall target="yaps-verify"/> <antcall target="barkbank-verify"/> <antcall target="petex-verify"/> </target> </pre>
Vérifie l'application YAPS Pet Store.	<pre> <target name="yaps-verify"> <mkdir dir="\${verifier.dir}"/> <exec executable="\${verifier}"> <arg line=" -d \${verifier.dir}"/> <arg line=" \${yaps.ear}"/> </exec> </target> </pre>
Vérifie l'application BarkBank.	<pre> <target name="barkbank-verify"> <mkdir dir="\${verifier.dir}"/> <exec executable="\${verifier}"> <arg line=" -d \${verifier.dir}"/> <arg line=" \${barkbank.war}"/> </exec> </target> </pre>
Vérifie l'application PetEx.	<pre> <target name="petex-verify"> <mkdir dir="\${verifier.dir}"/> <exec executable="\${verifier}"> <arg line=" -d \${verifier.dir}"/> <arg line=" \${petex.war}"/> </exec> </target> </pre>
Affiche les composants déployés sur le serveur GlassFish.	<pre> <target name="show"> <exec executable="\${asadmin}"> <arg line="show-component-status"/> <arg line=" --echo=\${echo}"/> <arg line=" --user \${server.user.name}"/> <arg line=" --passwordfile \${server.passwordfile}"/> <arg line=" --host \${server.host}"/> <arg line=" --port \${server.admin.port}"/> <arg line=" \${application.name}"/> </exec> </target> </pre>
Exécute l'application cliente.	<pre> <target name="run-client" depends="yaps-compile,yaps-build-client-jar"> <echo message="Runs the client application"/> <java classname="com.yaps.petstore.client.ui.PetstoreFrame" fork="yes" dir=". "> <classpath> <pathelement location="\${yaps.client.jar}"/> <path refid="classpath"/> </classpath> </java> </target> </pre>

```

<target name="db-insert-data">
  <sql src="${yaps.config.dir}/data.sql"
    driver="${db.driver}"
    url="${db.url}"
    userid="${db.user}"
    password="${db.password}">
</classpath refid="classpath"/>
</sql>
</target>

<target name="core" depends="clean,build"/>

</project>

```

◀ Insère des données dans la base.

◀ La tâche principale supprime les répertoires temporaires, compile toutes les applications et les package.

Admin.xml

Dans ce fichier, vous trouverez les tâches d'administration du serveur GlassFish. La plupart utilise la commande `asadmin` de GlassFish. Pour les exécuter, il suffit de taper la ligne de commande suivante :

```
ant -f admin.xml <le nom de la tâche>
```

Vous trouverez dans le fichier ci-dessous la liste des tâches d'administration que vous pouvez exécuter.

```

<?xml version="1.0"?>
<project name="Admin Petstore" default="list" basedir=".">
  <property name="application.name" value="petstore"/>
  <property name="home.dir" value="${basedir}"/>
  <property name="lib.dir" value="${home.dir}/lib"/>

  <property environment="env"/>
  <property name="glassfish.home"
    value="${env.GLASSFISH_HOME}"/>
  <property name="asadmin"
    value="${glassfish.home}/bin/asadmin.bat"/>
  <property name="echo" value="false"/>

  <property name="server.host" value="localhost"/>
  <property name="server.port" value="8080"/>
  <property name="server.admin.port" value="8282"/>
  <property name="server.jms.port" value="7676"/>
  <property name="server.user.name" value="admin"/>
  <property name="server.passwordfile" value="passwordfile"/>

  <property name="db.home" value="${glassfish.home}/javadb"/>
  <property name="db.lib" value="${db.home}/lib"/>
  <property name="db.host" value="localhost"/>
  <property name="db.port" value="1527"/>

```

◀ Variables globales de l'application.

◀ Variables d'environnement et utilitaires d'administration et de génération de GlassFish.

◀ Propriétés du serveur GlassFish.

◀ Propriétés de la base de données Derby.

GLASSFISH Administration

Pour obtenir de la documentation sur l'administration de GlassFish :

▶ <http://docs.sun.com/app/docs/doc/819-3658>

Propriétés de la source de données et du pool JDBC.

```
<property name="db.sid" value="${application.name}DB"/>
<property name="db.user" value="dbuser"/>
<property name="db.password" value="dbpwd"/>
<property name="db.datasource"
    value="org.apache.derby.jdbc.ClientDataSource"/>
<property name="db.driver"
    value="org.apache.derby.jdbc.ClientDriver"/>
<property name="db.url"
    value="jdbc:derby://${db.host}:${db.port}/${db.sid}"/>
<property name="db.schema.file" value="${db.sid}.schema"/>
```

Propriétés JMS.

```
<property name="jdbc.pool.name"
    value="${application.name}Pool"/>
<property name="jdbc.datasource.name"
    value="jdbc/${application.name}DS"/>

<property name="imq.home" value="${glassfish.home}/imq"/>
<property name="imq.lib" value="${imq.home}/lib"/>
<property name="jms.connection.factory.name"
    value="jms/${application.name}ConnectionFactory"/>
<property name="jms.topic" value="jms/topic/order"/>
```

JAR utilisés pour administrer l'application.

```
<path id="classpath">
    <pathelement location="${db.lib}/derbytools.jar"/>
    <pathelement location="${db.lib}/derbyclient.jar"/>
    <pathelement location="${db.lib}/derby.jar"/>
</path>
```

Démarrage le serveur GlassFish.

```
<target name="start-domain">
    <echo message="Starting ${application.name} domain for
        ${glassfish.home}"/>
    <exec executable="${asadmin}" failonerror="true"
        dir="${glassfish.home}">
        <arg line="start-domain"/>
        <arg line="--echo=${echo}"/>
        <arg line="${application.name}"/>
    </exec>
</target>
```

Arrête le serveur GlassFish.

```
<target name="stop-domain">
    <echo message="Stopping ${application.name} domain for
        ${glassfish.home}"/>
    <exec executable="${asadmin}" failonerror="true"
        dir="${glassfish.home}">
        <arg line="stop-domain"/>
        <arg line="--echo=${echo}"/>
        <arg line="${application.name}"/>
    </exec>
</target>
```

```

<target name="delete-domain">
  <echo message="Deleting ${application.name} domain for
                                ${glassfish.home}"/>
  <exec executable="${asadmin}" failonerror="true"
                                dir="${glassfish.home}">
    <arg line=" delete-domain"/>
    <arg line=" --echo=${echo}"/>
    <arg line=" ${application.name}"/>
  </exec>
</target>

<target name="start-db">
  <exec executable="${asadmin}" failonerror="true"
                                dir="${glassfish.home}">
    <arg value="start-database"/>
    <arg line=" --echo=${echo}"/>
    <arg value="--dbhost=${db.host}"/>
    <arg value="--dbport=${db.port}"/>
    <arg value="--dbhome=${db.home}"/>
  </exec>
</target>

<target name="stop-db">
  <exec executable="${asadmin}" failonerror="true"
                                dir="${glassfish.home}">
    <arg value="stop-database"/>
    <arg line=" --echo=${echo}"/>
    <arg value="--dbhost=${db.host}"/>
    <arg value="--dbport=${db.port}"/>
  </exec>
</target>

<target name="verify-db">
  <java classname="org.apache.derby.tools.sysinfo" fork="yes"
                                dir=". ">
    <classpath refid="classpath"/>
  </java>
</target>

<target name="create-connection-pool">
  <exec executable="${asadmin}">
    <arg line="create-jdbc-connection-pool"/>
    <arg line=" --echo=${echo}"/>
    <arg line=" --user ${server.user.name}"/>
    <arg line=" --passwordfile ${server.passwordfile}"/>
    <arg line=" --host ${server.host}"/>
    <arg line=" --port ${server.admin.port}"/>
    <arg line=" --datasourceclassname ${db.datasource}"/>
    <arg line=" --restype javax.sql.XADataSource"/>
    <arg line=" --property
      portNumber=${db.port}:serverName=${server.host}:
      User=${db.user}:Password=${db.password}:
      databaseName=${db.sid}:
      connectionAttributes=\;create\=true"/>
    <arg line=" ${jdbc.pool.name}"/>
  </exec>
</target>

```

◀ Supprime le domaine GlassFish.

◀ Démarre la base de données Derby.

◀ Arrête la base de données Derby.

◀ Vérifie la base de données Derby.

◀ Crée le pool de connexions.

Crée la source de données.

```
> <target name="create-datasource">
  <exec executable="${asadmin}">
    <arg line="create-jdbc-resource"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="--connectionpoolid ${jdbc.pool.name}"/>
    <arg line="--enabled=true"/>
    <arg line="${jdbc.datasource.name}"/>
  </exec>
</target>
```

Ping le pool de connexions.

```
> <target name="ping-connection-pool">
  <exec executable="${asadmin}">
    <arg line="ping-connection-pool"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="${jdbc.pool.name}"/>
  </exec>
</target>
```

Affiche la liste des pools de connexions.

```
> <target name="list-connection-pool">
  <exec executable="${asadmin}">
    <arg line="list-jdbc-connection-pools"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
  </exec>
</target>
```

Affiche la liste des sources de données.

```
> <target name="list-datasource">
  <exec executable="${asadmin}">
    <arg line="list-jdbc-resources"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
  </exec>
</target>
```

```
<target name="delete-connection-pool">
  <exec executable="${asadmin}">
    <arg line="delete-jdbc-connection-pool"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="${jdbc.pool.name}"/>
  </exec>
</target>
```

◀ Supprime le pool de connexions.

```
<target name="delete-datasource">
  <exec executable="${asadmin}">
    <arg line="delete-jdbc-resource"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="${jdbc.datasource.name}"/>
  </exec>
</target>
```

◀ Supprime la source de données.

```
<target name="list-jndi-resources">
  <exec executable="${asadmin}">
    <arg line="list-jndi-resources"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
  </exec>
</target>
```

◀ Affiche toutes les ressources JNDI.

```
<target name="list-jndi">
  <exec executable="${asadmin}">
    <arg line="list-jndi-entries"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
  </exec>
</target>
```

◀ Affiche l'arborescence JNDI.

Crée la fabrique de connexions JMS.

```
<target name="create-jms-connection-factory">
  <exec executable="${asadmin}">
    <arg line="create-jms-resource"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="--restype javax.jms.ConnectionFactory"/>
    <arg line="--enabled=true"/>
    <arg line="${jms.connection.factory.name}"/>
  </exec>
</target>
```

Crée le Topic JMS.

```
<target name="create-jms-topic">
  <exec executable="${asadmin}">
    <arg line="create-jms-resource"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="--restype javax.jms.Topic"/>
    <arg line="--enabled=true"/>
    <arg line="--property Name=OrderTopic"/>
    <arg line="${jms.topic}"/>
  </exec>
</target>
```

Affiche les ressources JMS.

```
<target name="list-jms-resources">
  <exec executable="${asadmin}">
    <arg line="list-jms-resources"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
  </exec>
</target>
```

Supprime la fabrique de connexions JMS.

```
<target name="delete-jms-connection-factory">
  <exec executable="${asadmin}">
    <arg line="delete-jms-resource"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="${jms.connection.factory.name}"/>
  </exec>
</target>
```

```

<target name="set-loggers">
  <exec executable="${asadmin}">
    <arg line="set"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="server.log-service.
      module-log-levels.property.com\\.yaps\\.petstore=FINEST "/>
  </exec>

  <exec executable="${asadmin}">
    <arg line="set"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="server.log-service.
      module-log-levels.property.com\\.barkbank=FINEST "/>
  </exec>

  <exec executable="${asadmin}">
    <arg line="set"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="server.log-service.
      module-log-levels.property.com\\.petex=FINEST "/>
  </exec>
</target>

<target name="version">
  <exec executable="${asadmin}">
    <arg line="version"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
    <arg line="--verbose=true"/>
  </exec>
</target>

```

◀ Crée les loggers pour les applications YAPS
Pet SStore, BarkBank, PetEx.

◀ Affiche la version du serveur GlassFish.

Affiche la liste des composants déployés sur GlassFish.

```
> <target name="list-components">
  <exec executable="${asadmin}">
    <arg line="list-components"/>
    <arg line="--echo=${echo}"/>
    <arg line="--user ${server.user.name}"/>
    <arg line="--passwordfile ${server.passwordfile}"/>
    <arg line="--host ${server.host}"/>
    <arg line="--port ${server.admin.port}"/>
  </exec>
</target>
```

Affiche la totalité des informations du serveur GlassFish.

```
> <target name="list" description="list all components">
  <antcall target="list-connection-pool"/>
  <antcall target="list-datasource"/>
  <antcall target="list-jms-resources"/>
  <antcall target="list-jndi"/>
  <antcall target="list-components"/>
</target>
```

Configure le serveur GlassFish.

```
> <target name="setup">
  <antcall target="create-connection-pool"/>
  <antcall target="ping-connection-pool"/>
  <antcall target="create-datasource"/>
  <antcall target="create-jms-connection-factory"/>
  <antcall target="create-jms-topic"/>
  <antcall target="set-loggers"/>
</target>

</project>
```

Sigles et acronymes



Le monde de l’informatique, et plus particulièrement Java, est parsemé d’une multitude de sigles et d’acronymes en tout genre. Vous trouverez ci-après une liste non exhaustive de ceux rencontrés les plus fréquemment dans la littérature informatique.

Tableau C-1 Sigles et acronymes

Sigle/Acronyme	Signification
ACC	Application Client Container
Ant	Another Neat Tool
API	Application Programming Interface
ASP	Active Server Page
AWT	Abstract Windowing Toolkit
B2A	Business to Administration
B2B	Business to Business
B2C	Business to Consumer
BLOB	Binary Large Object
BMP	Bean-Managed Persistence
C2C	Consumer to Consumer
CGI	Common Gateway Interface
CMP	Container-Managed Persistence
CMR	Container-Managed Relation
CMT	Container-Managed Transaction
CRUD	Create/Read/Update/Delete
DAO	Data Access Object
DDL	Data Definition Language

Tableau C–1 Sigles et acronymes (suite)

Sigle/Acronyme	Signification
DOM	Document Object Model
DTD	Document Type Definition
DTO	Data Transfer Object
EAR	Enterprise ARchive
EJB	Enterprise Java Bean
EJBQL	Enterprise Java Bean Query Language
EL	Expression Language
FCS	First Customer Shipment
GoF	Gang of Four
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure (sockets)
I18n	Internationalization
IDE	Integrated Development Environment
IoC	Inversion of Control
IP	Internet Protocol
JAAS	Java Authentication and Authorization Service
JACC	Java Authorization Contract for Containers
JAF	JavaBeans Activation Framework
JAR	Java ARchive
JAX-RPC	Java API for XML-based RPC
JAX-WS	Java API for XML-based Web Services
JAXB	Java Architecture for XML Binding
JCA	Java Connector Architecture
JDBC	Java DataBase Connectivity
JDK	Java Development Kit
JDO	Java Data Object
Java EE (ou JEE)	Java Enterprise Edition
JMS	Java Message Service
JNDI	Java Naming Directory Interface
JPA	Java Persistence API
JRE	Java Runtime Environment
JRMP	Java Remote Method Protocol
Java SE (ou JSE)	Java Standard Edition
JSF	JavaServer Faces
JSP	JavaServer Pages

Tableau C-1 Sigles et acronymes (suite)

Sigle/Acronyme	Signification
JSR	Java Specification Request
JSTL	JSP Standard Tag Library
JTA	Java Transaction API
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MIME	Multipurpose Internet Mail Extensions
MOM	Message-Oriented Middleware
ODBC	Open DataBase Connectivity
OMG	Object Management Group
PDA	Personal Digital Assistant
PHP	PHP Hypertext Preprocessor
POJO	Plain Old Java Object
RC	Release Candidate
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAAJ	SOAP with Attachments API for Java
SAX	Simple API for XML
SGML	Standard Generalized Markup Language
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSL	Secure Socket Layer
SSO	Single Sign-On
StAX	Streaming API for XML
TCP	Transmission Control Protocol
UDDI	Universal Description, Discovery and Integration
UDP	User Datagram Protocol
UEL	Unified Expression Language
UML	Unified Modeling Language
UR	Update Release
URC	Uniform Resource Characteristic
URI	Unified Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
WAP	Wireless Application Protocol
WAR	Web ARchive
WML	Wireless Markup Language

Tableau C–1 Sigles et acronymes (suite)

Sigle/Acronyme	Signification
WSDL	Web Service Description Language
XML	eXtended Markup Language
XSL	eXtensible Stylesheet Language
XSLT	eXtensible Stylesheet Language Transformations
YAPS	Yet Another PetStore

EJB 2

D

Entity Bean et Entity

Les EJB 2.x utilisent le terme d'*entity bean* pour définir les composants persistants. Avec JPA, on parle plutôt tout simplement d'*entity* (ou entité).

Outils Base de données MySQL

MySQL est un serveur de base de données relationnelle, multithreadé, multi-utilisateurs et robuste. Il dispose de deux licences : Open Source, sous les termes de la licence GNU GPL (*General Public License*), ou alors une licence commerciale achetée auprès de MySQL AB.

► <http://www.mysql.com>

Outils Serveur d'applications JBoss

JBoss est un serveur d'applications Open Source certifié J2EE. Historiquement spécialisé pour les EJB, son architecture modulaire lui permet de répondre aux autres spécifications J2EE : JBoss Server est le conteneur d'EJB, JBoss MQ gère les messages (JMS), JBoss MX la messagerie électronique, JBoss TX les transactions JTA/JTS, JBoss SX la sécurité, JBoss CX la connectivité JCA et JBoss CMP la persistance CMP. Pour les JSP/servlets, JBoss intègre Tomcat.

► <http://www.jboss.com/>

« Java EE 5 est plus simple que J2EE 1.4 ». Voilà une phrase qui revient assez souvent dans ce livre. Pour vous le prouver, cette annexe se focalise sur les EJB et se propose de vous expliquer comment développer un stateless bean et un entity bean en version 2.1. Nous prendrons deux exemples extrêmement simples de type « Hello World » qui vous permettront de comprendre les différences entre les EJB 2 et les EJB 3.

Pour changer un peu de configuration, ces EJB utilisent des fichiers de déploiement pour le serveur d'applications JBoss et la base MySQL.

Un exemple d'entity bean

Les entity beans 2.x se déclinent en deux modèles de persistance :

- au niveau du composant (BMP : *Bean-Managed Persistence*) ;
- au niveau du conteneur (CMP : *Container-Managed Persistence*).

La persistance d'un BMP est implémentée par le développeur. Ce dernier a en charge le développement de la logique nécessaire pour insérer les données en base, les mettre à jour, les supprimer et instancier un EJB à partir de ces données. Cela nécessite habituellement d'écrire du code JDBC. Avec cette gestion de persistance, le développeur a un contrôle total sur la manière dont les accès à la base sont réalisés.

La persistance d'un CMP est assurée par le conteneur d'EJB. À travers une configuration spécifiée dans un descripteur de déploiement, le conteneur d'EJB fera correspondre les attributs de l'entity bean avec les colonnes d'une table. L'utilisation des CMP réduit le temps de développement ainsi que le code, mais augmente la complexité en utilisant des fichiers XML.

L'interface de fabrique hérite de <code>javax.ejb.EJBHome</code> .	▶
Cette méthode permet d'instancier un entity bean et d'insérer ses données en base.	▶
Permet de retrouver un entity bean à partir d'un paramètre (ici la clé). Le type de retour est l'interface métier.	▶

EJB Remote et Local

Les entity beans 2.1 peuvent être invoqués de manière distante ou locale. Selon le cas, l'interface de fabrique hérite soit de `EJBHome`, soit de `EJBLocalHome` et l'interface métier de `EJBObject` ou de `EJBLocalObject`. Pour les appels locaux, les méthodes ne lancent pas de `RemoteException`.

L'interface métier hérite de <code>javax.ejb.EJBObject</code> .	▶
Accesseurs des deux attributs de l'entity bean clé et valeur.	▶

L'exemple ci-après utilise un CMP qui persiste des clés et des valeurs dans une table constituée de deux colonnes.

Les entity beans 2.x sont constitués de deux interfaces et d'une classe. L'interface de fabrique (home interface) définit les méthodes qu'un client peut invoquer pour créer, trouver ou supprimer un EJB. Étant donné que les EJB entity sont persistants, un client peut créer une instance d'un composant (c'est-à-dire faire un insert dans la table) mais aussi en rechercher une existante (un ordre select). Notez que l'invocation de ces méthodes peut échouer, soit pour des raisons liées au réseau (`RemoteException`), soit pour des problèmes d'accès à la base de données (`CreateException`, `FinderException`).

Home interface de l'entity bean

```
public interface HelloHome extends EJBHome {  
  
    Hello create(String cle) throws RemoteException,  
                                                CreateException;  
  
    Hello findByPrimaryKey(String cle) throws RemoteException,  
                                                FinderException;  
}
```

L'interface métier permet de définir les services que propose le composant. Elle définit les accesseurs des attributs du composant ainsi que d'éventuelles méthodes métiers. Dans notre cas, sont définis les accesseurs des attributs clé et valeur. Notez que l'invocation de ces méthodes peut échouer pour des raisons liées au réseau (les méthodes lancent des `RemoteException`) ; nous manions par conséquent ce que l'on appelle un composant distant.

Interface métier de l'entity bean

```
public interface Hello extends EJBObject {  
  
    String getCle() throws RemoteException;  
    String getValeur() throws RemoteException;  
    void setValeur(String valeur) throws RemoteException;  
}
```

La classe du bean n'implémente pas les interfaces que nous venons de définir, mais plutôt l'interface `EntityBean` ❶. Notez qu'un CMP ne possède pas d'attributs et que les accesseurs sont déclarés `abstract` ❷. La signature de la méthode `ejbCreate()` doit être la même que `ejbPostCreate` ❸ et correspond à la méthode `create()` déclarée dans l'interface de fabrique. Les méthodes permettant de gérer le cycle de vie du composant sont vides ❹.

Classe d'implémentation de l'entity bean

```
public abstract class HelloBean implements EntityBean { ❶
    public abstract String getCle() throws RemoteException;
    public abstract void setCle(String cle) throws ❷
                                                RemoteException;
    public abstract String getValeur() throws RemoteException;
    public abstract void setValeur(String valeur) throws
                                                RemoteException;
    public String ejbCreate(String cle) throws RemoteException,
                                                CreateException {
        setCle(cle);
        return null;
    }

    public void ejbPostCreate(String cle) throws CreateException{
    } ❸

    public void setEntityContext(EntityContext entityContext)
                                throws EJBException { }
    public void unsetEntityContext() throws EJBException { }
    public void ejbRemove() throws RemoveException,
                                EJBException { }
    public void ejbActivate() throws EJBException { }
    public void ejbPassivate() throws EJBException { }
    public void ejbload() throws EJBException { } ❹
    public void ejbStore() throws EJBException { }
}
```

- ◀ La classe de l'entity bean est abstraite.
- ◀ Les accesseurs sont abstraits et lancent une `RemoteException`.
- ◀ Chaque méthode create de l'interface de fabrique doit avoir une méthode `ejbCreate` dans la classe du bean. Notez que cette méthode retourne la valeur `null`.
- ◀ Une méthode `ejbCreate` doit avoir obligatoirement une méthode `ejbPostCreate` qui permet d'exécuter du code après avoir inséré des données en base.
- ◀ Ces méthodes de callback doivent être implémentées même si on ne les utilise pas.

Comme vous pouvez le constater, il n'y a ni ordres SQL, ni attributs, ni colonnes ou tables dans lesquelles les données doivent persister. La plupart de ces informations sont décrites dans le fichier normé `ejb-jar.xml` ainsi que dans des fichiers spécifiques au conteneur d'EJB (`jbosscmp-jdbc.xml` pour JBoss).

Dans le fichier `ejb-jar.xml`, on déclare le nom de l'entity bean ❺ ainsi que les interfaces ❻ et classes d'implémentation ❼. La balise `persistence-type` ❽ avertit le conteneur qu'il doit gérer la persistance (Container pour les CMP ou Bean pour les BMP). La clé primaire que nous utilisons est de type `String` ❾. Dans les balises `cmp-field` ❿, on déclare les attributs (`cle` et `valeur`) du composant ⓫ puis on indique au conteneur lequel de ces attributs est la clé primaire ⓬.

Chaque entity bean 2.x doit être défini dans le fichier `ejb-jar.xml`.

Les CMP délèguent la persistance au conteneur.

Tous les attributs persistants doivent être répertoriés dans cette section.

La méthode `findByPrimaryKey` s'appuie sur cette information.

Fichier `ejb-jar.xml`

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <display-name>HelloEB</display-name>
      <ejb-name>HelloBean</ejb-name> ⑤
      <home>HelloHome</home> ⑥
      <remote>Hello</remote> ⑥
      <ejb-class>HelloBean</ejb-class> ⑦
      <persistence-type>Container</persistence-type> ⑧
      <prim-key-class>java.lang.String</prim-key-class> ⑨
      <reentrant>False</reentrant>
      <cmp-version>2.x</cmp-version>
      <abstract-schema-name>Hello</abstract-schema-name>;
      <cmp-field> ⑩
        <field-name>cle</field-name> ⑪
      </cmp-field>
      <cmp-field>
        <field-name>valeur</field-name> ⑪
      </cmp-field>
      <primkey-field>cle</primkey-field> ⑫
    </entity>
  </enterprise-beans>
</ejb-jar>
```

Le fichier `jbosscmp-jdbc.xml` effectue le mapping entre le composant et la base de données. Tout d'abord, ce fichier déclare la source de données `petstoreDS` ⑬ que l'on va utiliser ainsi qu'un mapping propre à la base de données `MySQL` ⑭. On définit la table où les données vont être stockées ⑮, dans notre cas `HELLO_PETSTORE`, puis le mapping entre les attributs du composant et les colonnes de la table ⑯.

Fichier `jbosscmp-jdbc.xml`

Source de données et type de base de données utilisés pour la persistance.

L'entity bean défini sous le nom `HelloBean` dans le fichier `ejb-jar.xml` est persisté dans la table `Hello_Petstore`.

```
<jbosscmp-jdbc>
  <defaults>
    <datasource>java:/petstoreDS</datasource> ⑬
    <datasource-mapping>mysql</datasource-mapping> ⑭
  </defaults>
  <enterprise-beans>
    <entity>
      <ejb-name>HelloBean</ejb-name>
      <table-name>HELLO_PETSTORE</table-name> ⑮
    </entity>
  </enterprise-beans>
</jbosscmp-jdbc>
```

```

<cmp-field> ⑩
  <field-name>cle</field-name>
  <column-name>key</column-name>
  <jdbc-type>VARCHAR</jdbc-type>
  <sql-type>varchar(10)</sql-type>
</cmp-field>

<cmp-field> ⑩
  <field-name>valeur</field-name>
  <column-name>value</column-name>
  <jdbc-type>VARCHAR</jdbc-type>
  <sql-type>varchar(50)</sql-type>
</cmp-field>
</entity>
</enterprise-beans>
</jbosscmp-jdbc>

```

◀ L'attribut clé est stocké dans la colonne key de type varchar(10).

◀ L'attribut valeur est stocké dans la colonne value de type varchar(50).

Il faut ensuite compiler et packager tous ces fichiers dans un JAR pour pouvoir déployer l'EJB. Il reste maintenant à développer une classe cliente qui pourra interagir avec ce composant. Il n'y a pas d'injection en EJB 2.x, il faut donc obligatoirement utiliser JNDI pour retrouver l'interface de l'entity bean ② à partir du contexte initial ①. Dans cet exemple, on insère ③ des données en base à l'aide de la méthode create(). La méthode findByPrimaryKey nous permet de retrouver le composant ④ et d'en modifier les valeurs ⑤. Pour supprimer les données de la base, on appelle la méthode remove ⑥ de l'entity bean.

Classe utilisant l'entity bean

```

public class Main {

    public static void main(String[] args) {
        InitialContext ic = null;
        try {
            Properties props = new Properties();
            props.setProperty("java.naming.factory.initial",
                "org.jnp.interfaces.NamingContextFactory");
            props.setProperty("java.naming.factory.url.pkgs",
                "org.jboss.naming:org.jnp.interfaces");
            props.setProperty("java.naming.provider.url",
                "localhost");

            ic = new InitialContext(props); ①
            Object objRef = (HelloHome) ic.lookup("ejb/Hello"); ②
            HelloHome home = (HelloHome)
                PortableRemoteObject.narrow(objRef, HelloHome.class);
            Hello hello = home.create("Hello"); ③
            hello.setValeur("PetStore!");

            hello = home.findByPrimaryKey("Hello"); ④
            System.out.println(hello.getCle());
            System.out.println(hello.getValeur());

```

◀ Propriétés pour accéder à l'annuaire JNDI de JBoss.

◀ On recherche dans JNDI l'interface métier qui se nomme ejb/Hello.

◀ On crée un entity bean avec les attributs « Hello » pour la clé et « PetStore! » pour la valeur.

◀ Une fois créé, on le recherche à partir de sa clé primaire. Une fois obtenu, on affiche la valeur des attributs.

En modifiant les attributs, on met à jour la base de données.

L'entity bean est supprimé ainsi que ses données en base.

```

        hello.setValeur("PetStore Modifié!"); ❸

        hello.remove(); ❹

    } catch (Exception e) {
        e.printStackTrace();
    }
}

```

Comme vous pouvez le constater, les entity beans 2.x sont bien différents des Pojo annotés par JPA. Ils utilisent le système d'interface de fabrique (qui rappelle l'entity manager) et sont en réalité des classes abstraites sans attributs. De plus, les informations de mapping sont éparpillées dans plusieurs fichiers XML, ce qui rend les éventuelles erreurs difficiles à trouver.

Un exemple de stateless bean

Pour le stateless bean, que diriez-vous d'un composant qui retourne la chaîne de caractère « Hello Petstore! » ainsi que la date du jour. Pour développer ce stateless session bean 2.x, il faut deux interfaces, une classe et un descripteur de déploiement.

Une première interface de fabrique permet la construction (factory) d'un composant EJB. Dans notre cas, il s'agit d'une interface distante étant donné qu'EJBHome ❶ hérite de l'interface `java.rmi.Remote`. Le nom d'une méthode de création est obligatoirement `create` ❷.

Home interface du stateless bean

L'interface de fabrique hérite de `javax.ejb.EJBHome`.

```

public interface HelloHome extends EJBHome { ❶
    Hello create() throws RemoteException, CreateException; ❷
}

```

L'interface métier permet de définir les services que propose l'EJB, c'est-à-dire les méthodes métiers qu'expose le composant (dans notre cas les deux méthodes `sayHello()` et `today()`). Notez que l'invocation de ces méthodes peut échouer pour des raisons liées au réseau (`RemoteException`).

Interface métier du stateless bean

L'interface métier hérite de `javax.ejb.EJBObject`.

Les méthodes métiers permettant de retourner une chaîne de caractères et la date du jour.

```

public interface Hello extends EJBObject {

    String sayHello() throws RemoteException;
    Date today() throws RemoteException;

}

```

Contrairement aux EJB 3, la classe d'implémentation des EJB 2 n'implémente pas l'interface métier. Notez aussi que les deux méthodes métiers `sayHello` ③ et `today` ④ ne lancent pas de `RemoteException` alors qu'elles sont déclarées comme telles dans l'interface métier. Les méthodes permettant de gérer le cycle de vie du composant doivent être implémentées même si elles ne sont pas utilisées ⑤.

Classe d'implémentation du stateless bean

```
public class HelloBean implements SessionBean {

    public HelloBean() { }

    public String sayHello() {
        return "Hello Petstore !"; ③
    }

    public Date today() {
        return new Date(); ④
    }

    public void ejbCreate() throws CreateException { } ⑤
    public void setSessionContext(SessionContext sessionContext)
        throws EJBException { }
    public void ejbRemove() throws EJBException { }
    public void ejbActivate() throws EJBException { }
    public void ejbPassivate() throws EJBException { }
}
```

- ◀ Un EJB stateless 2.x hérite de `javax.ejb.SessionBean`.
- ◀ Constructeur.
- ◀ Les méthodes métier.
- ◀ Les méthodes callback.

Tout comme Java EE, on peut démarquer les transactions dans les EJB stateless en tenant compte d'une politique transactionnelle qui peut être gérée soit manuellement, soit par le conteneur ⑧. Les annotations n'existant pas (`@TransactionAttribute` en EJB 3), la politique transactionnelle est spécifiée ⑨ dans le descripteur de déploiement XML (fichier `ejb-jar.xml`). Ce fichier informe aussi le conteneur du nom des interfaces, de la classe métier ⑥ et aussi du type de composant ⑦, dans notre cas un composant session sans état.

Fichier `ejb-jar.xml`

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <display-name>HelloSB</display-name>
      <ejb-name>HelloBean</ejb-name>
      <home>HelloHome</home>
      <remote>Hello</remote> ⑥
      <ejb-class>HelloBean</ejb-class>
```

- ◀ Nom de l'EJB.
- ◀ Nom de l'interface de fabrique, de l'interface métier et de la classe d'implémentation.

L'EJB est stateless et la démarcation des transactions est assurée par le conteneur.

La politique transactionnelle utilisée pour toutes les méthodes de ce session bean est Required.

```
<session-type>Stateless</session-type> 7
<transaction-type>Container</transaction-type> 8

</session>

<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>HelloBean</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute> 9
  </container-transaction>
</assembly-descriptor>

</enterprise-beans>
</ejb-jar>
```

Un autre fichier de description propre au serveur d'applications (dans notre cas jboss.xml), permet de donner un nom JNDI à l'EJB afin qu'il puisse être appellable par un client. Le composant HelloBean 10 se nomme donc ejb/Hello 11.

Fichier jboss.xml

Nom JNDI du stateless session bean.

```
<jboss>
  <enterprise-beans>
    <session>
      <ejb-name>HelloBean</ejb-name> 10
      <jndi-name>ejb/Hello</jndi-name> 11
    </session>
  </enterprise-beans>
</jboss>
```

Il faut ensuite compiler les classes et les packager avec les descripteurs XML dans un JAR pour pouvoir déployer l'EJB. Pour interagir avec ce composant, il faut développer une classe cliente qui utilise JNDI pour retrouver l'interface du session bean 1. Une fois l'interface obtenue, on peut appeler les méthodes métier 2.

Classe utilisant le stateless bean

Propriétés pour accéder à l'annuaire JNDI de JBoss.

```
public class Main {

  public static void main(String[] args) {
    InitialContext ic = null;
    try {
      Properties props = new Properties();
      props.setProperty("java.naming.factory.initial",
        "org.jnp.interfaces.NamingContextFactory");
      props.setProperty("java.naming.factory.url.pkgs",
        "org.jboss.naming:org.jnp.interfaces");
      props.setProperty("java.naming.provider.url",
        "localhost");
```

```

    ic = new InitialContext(props); ❶
    Object objRef = (HelloHome) ic.lookup("ejb/Hello");
    HelloHome home = (HelloHome)PortableRemoteObject.narrow
        (objRef, HelloHome.class);
    Hello hello = home.create();
    System.out.println(hello.sayHello());
    System.out.println(hello.today()); ❷
} catch (Exception e) {
    e.printStackTrace();
}
}
}

```

◀ On recherche dans JNDI l'interface métier qui se nomme « ejb/Hello ».

◀ On appelle les deux méthodes métiers.

En résumé

La spécification 3 des EJB a été élaborée en vue de simplifier la conception d'EJB pour le développeur. Les avantages sont les suivants :

- une architecture basée sur les Pojos et non plus sur les composants ;
- une API pour le mapping objet/relationnel (JPA) ;
- moins de classes et d'interfaces à implémenter (plus besoin d'hériter d'une super interface ou classe) ;
- utilisation des annotations à la place des descripteurs de déploiement XML devenus optionnels ;
- avec le système d'injection, les JNDI lookups ne sont plus nécessaires ;
- une gestion du cycle de vie simplifiée.

Développement avec IntelliJ IDEA

E

RÉFÉRENCE **IntelliJ IDEA**

IntelliJ IDEA est un IDE créé par la société JetBrains.

► <http://www.jetbrains.com/idea/>

IntelliJ IDEA est un environnement de développement intégré (IDE) qui facilite le développement d'applications Java EE en vous fournissant un support étendu sur une grande variété de frameworks et de technologies.

IntelliJ IDEA vous offre un ensemble d'outils qui simplifient le processus de développement, en vous fournissant notamment une aide à la programmation avec complétion de code intelligente, des avertissements d'erreurs à la volée, des diagrammes d'entités-relations pour la modélisation, du remaniement de code (refactoring), de la génération automatique de code, etc.

Dans les projets où il est nécessaire d'employer de multiples technologies et frameworks, IntelliJ IDEA vous fournit un niveau de support inégalé en intégrant tous ces outils de manière cohérente. IntelliJ IDEA vous donne également la flexibilité de décrire la structure de votre application (via les annotations ou les descripteurs XML) afin de manipuler ce mélange de technologies de manière intégrée.

Les assistants au développement sont disponibles aussi bien pour Java que pour XML, SQL, JSP, HTML et tout autre code dont vous pourriez avoir besoin dans vos projets. IntelliJ IDEA détecte le langage dont vous avez besoin et vous fournit une aide adéquate. Par exemple, là où vous devez éditer un ordre SQL, vous aurez de la complétion de code SQL.

L'éditeur JSP (*Java Server Pages*) vous assiste pratiquement dans tout ce dont vous pourriez avoir besoin pour créer une interface utilisateur : il reconnaît les frameworks et les balises que vous utilisez, vérifie le code, répare les erreurs fréquentes et s'intègre avec divers outils comme le diagramme visuel JSF (*Java Server Faces*).

Cette annexe se propose de vous montrer comment développer une application web permettant à un utilisateur de modifier les données d'un client et de son adresse en utilisant Java Persistence API (JPA) et JSF.

Un projet façon IntelliJ IDEA

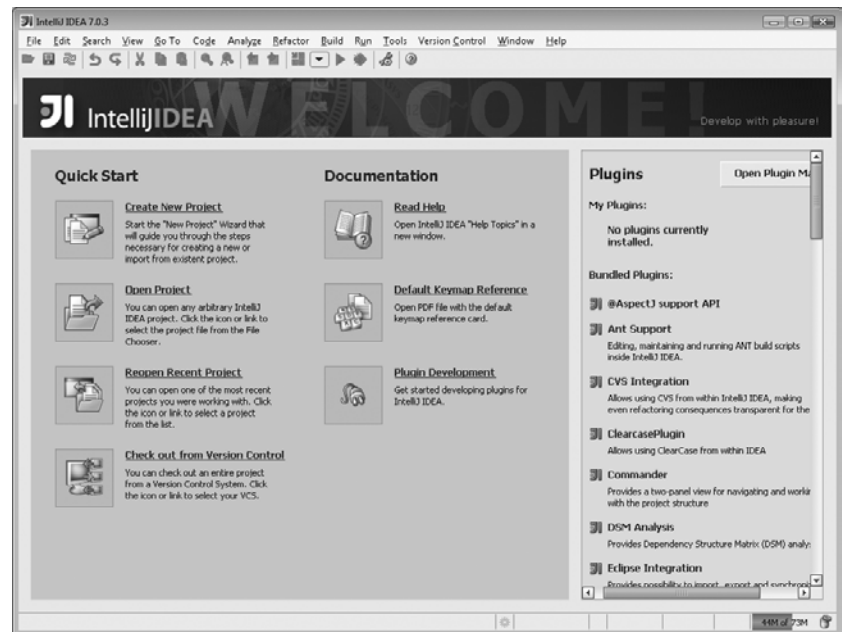
IntelliJ IDEA vous laisse contrôler la configuration de votre projet par l'emploi de « facettes ». Chaque facette supporte une technologie ou un framework particulier. Une fois qu'elle est ajoutée à votre projet, vous avez accès aux divers outils d'IntelliJ IDEA ainsi qu'aux éditeurs de code prêts à l'emploi.

Certaines facettes, comme JSF, ont besoin d'une facette parent (facette web). Dans ce cas, la facette web permet de supporter les servlets, les descripteurs de déploiement, la configuration XML, etc.

Créer et configurer le projet

Pour créer un projet, lancez IntelliJ IDEA et cliquez sur *Create New Project* sur l'écran de démarrage ou dans le menu *File>New Project* (figure E-1).

Figure E-1
Création d'un projet



L'assistant de projet apparaît alors. Il est possible de créer un projet à partir de code existant ou à partir de zéro. C'est cette dernière option

que nous allons choisir : sélectionnez *Create project from scratch* et cliquez sur *Next* (figure E-2).

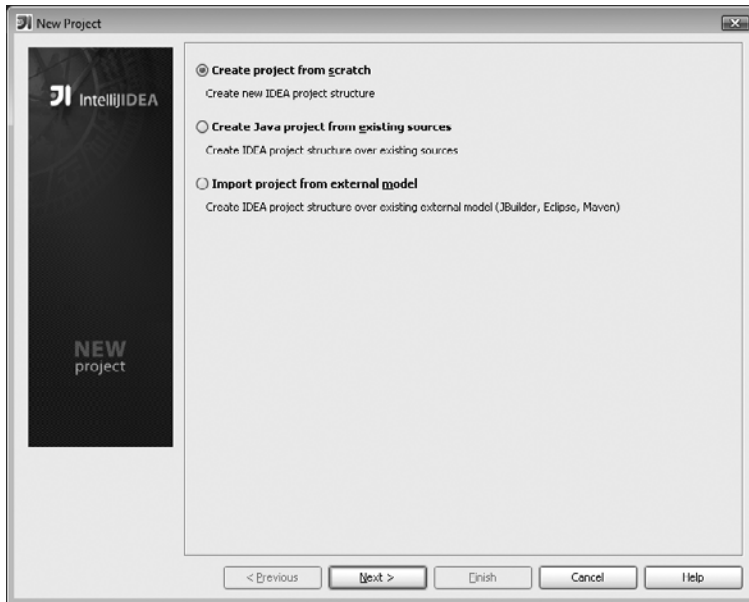


Figure E-2

Création d'un projet à partir de rien

Ici nous spécifions le nom du projet (*Yaps Customer Management*) et demandons à IntelliJ IDEA d'utiliser le module Java (*Java Module*) (figure E-3).

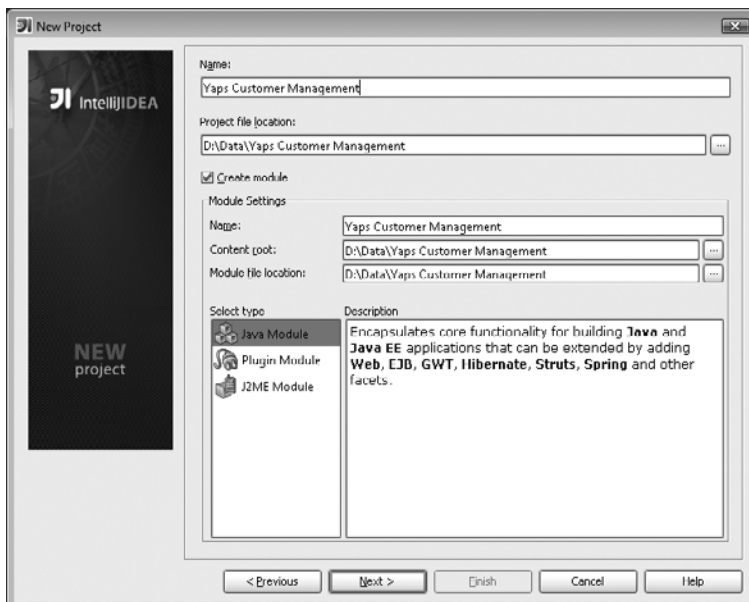


Figure E-3

Création du projet de gestion des clients

Choisissez le répertoire où sera stocké le projet ainsi que le code source (par défaut *src*), et cliquez sur *Next*. Vous arrivez sur un écran où il faut spécifier les technologies et frameworks utilisés dans le projet (figure E-4).

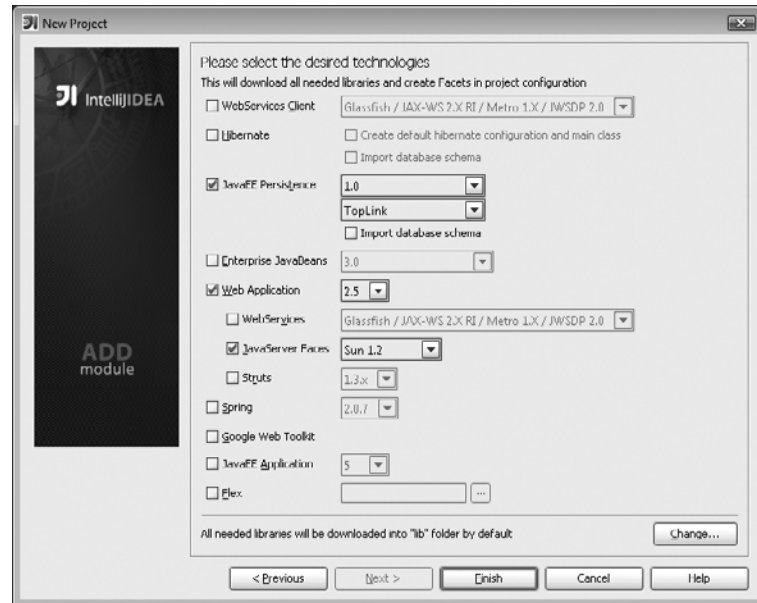


Figure E-4
Sélectionnez les technologies
utilisées par le projet.

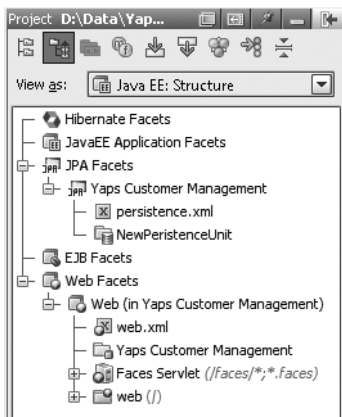


Figure E-5
Structure du projet

Pour la persistance Java EE (c'est-à-dire JPA), sélectionnez *TopLink* et, pour l'application web, *JSF Sun 1.2*. Cliquez enfin sur *Finish*. IntelliJ IDEA créera et configurera automatiquement le projet, les facettes, et téléchargera les bibliothèques nécessaires pour les ajouter dans le répertoire *lib* (répertoire par défaut).

Le projet est créé. Dans la fenêtre de gauche, vous pouvez changer le mode d'affichage (liste déroulante intitulée *View as*) pour voir la structure Java EE (choisissez *Java EE : Structure*) (figure E-5). Comme vous pouvez le constater, les fichiers de configuration web (*web.xml* et *faces-config.xml*) ont été créés, ainsi que ceux de la persistance (*persistence.xml*).

Il nous faut maintenant créer les éléments de l'application *Yaps Customer Management*.

Créer les éléments de l'application

Avant toute chose, il nous faut renommer l'unité de persistance par défaut *NewPersistenceUnit* en *petstorePU*. Pour cela, sélectionnez l'unité de persistance, pressez les touches *Shift+F6* et saisissez *petstorePU* (figure E-6).

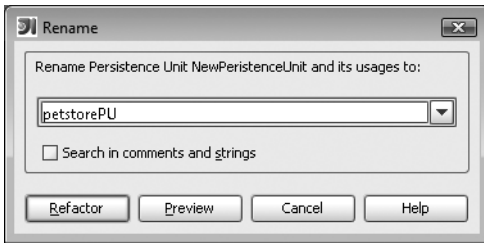


Figure E-6
Renommez l'unité de persistance.

Cliquez sur *Refactor*. Comme le projet ne contient pas encore de code, IntelliJ IDEA a simplement renommé l'unité de persistance. S'il y avait déjà eu des références à cette unité, elles auraient été mises à jour.

Il est temps désormais d'ajouter des entités à cette unité de persistance.

Créer l'entity Address

La création d'un entity est extrêmement simple : faites un clic droit sur l'unité de persistance `petstorePU`, sélectionnez le menu *New*, puis cliquez sur *Entity*. IntelliJ IDEA vous demande alors le nom de l'entité (saisissez `Address`) ainsi que son paquetage cible (`com.yaps.petstore.entity`).

L'entity ainsi créé, on peut rajouter des attributs en faisant un clic droit sur cet entity `Address` puis en sélectionnant le menu *New*. S'affiche alors la liste des attributs possibles.

Commençons par créer un identifiant (sélectionnez *New* puis *id*). IntelliJ IDEA affiche une boîte de dialogue nous permettant de spécifier le nom du champ (`id`) ainsi que son type (`java.lang.Long`) (figure E-7). Notez que vous pouvez appuyer sur *Ctrl+Espace* pour obtenir la complétion sur le type.

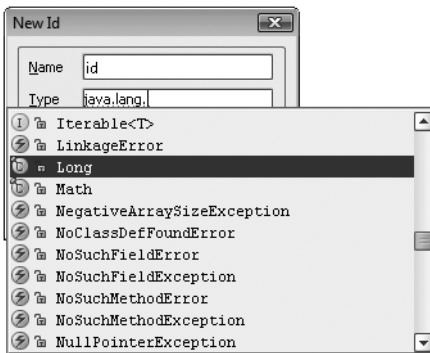


Figure E-7
Création de l'identifiant de l'adresse

Vous pouvez constater qu'IntelliJ IDEA a généré l'attribut `id` ainsi que les méthodes d'accès `get` et `set`.

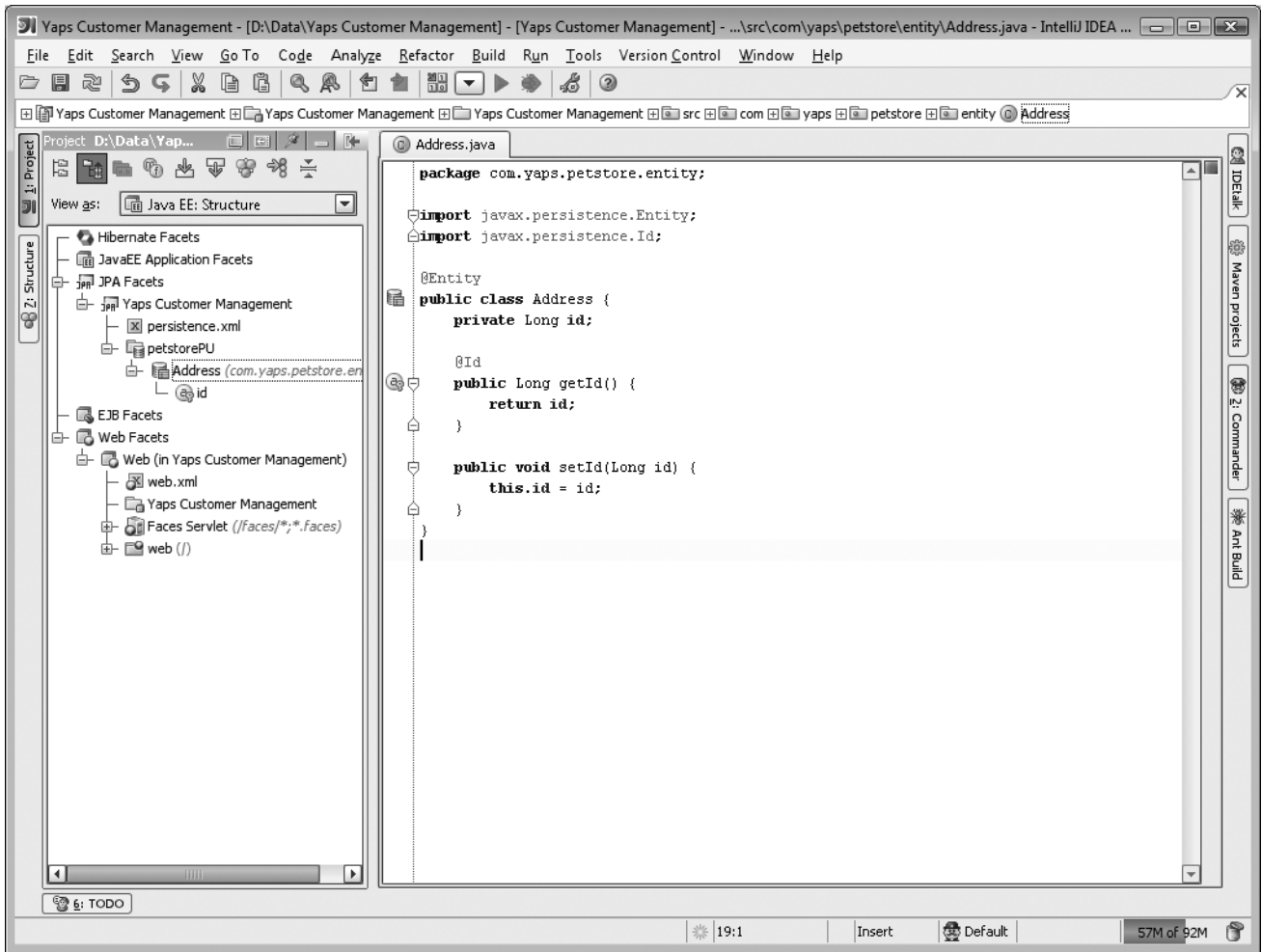


Figure E-8 Entity Address avec identifiant, getter et setter

```

@Entity
public class Address {
    private Long id;
    // javax.persistence.GeneratedValue? Alt+Entrée
    @Id @GeneratedValue
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}

```

Figure E-9 Import automatique de l'annotation @GeneratedValue

Cet identifiant doit être généré automatiquement. Pour cela, il suffit de l'annoter avec `@GeneratedValue`.

IntelliJ IDEA a surligné l'annotation car elle est encore inconnue. En revanche, il se propose de l'importer automatiquement (figure E-9). Pour cela, appuyez sur **Alt+Entrée** et vous verrez l'ajout de la ligne `import javax.persistence.GeneratedValue;` en haut de la classe.

Utilisez l'assistant pour créer les autres attributs de l'entity Address : `street1`, `street2`, `city`, `state`, `zipcode` et `country` ; ces attributs sont tous de type `String`.

Méthodes de callback

Maintenant que nous avons tous les attributs de l'entité `Address`, il nous faut ajouter les méthodes de callback pour nous assurer de la validité des données avant insertion (`@PrePersist`) et mise à jour (`@PreUpdate`). Appuyez sur les touches **Alt+Ins** et sélectionnez les méthodes de callback partir du menu *Entity Listener Methods* (figure E-10).

Saisissez le code de validation dans la méthode annotée `validateData`. Profitez en pour surcharger les méthodes `equals` et `hashCode` à partir du même menu *Generate* (touches **Alt+Ins**) (figure E-11).

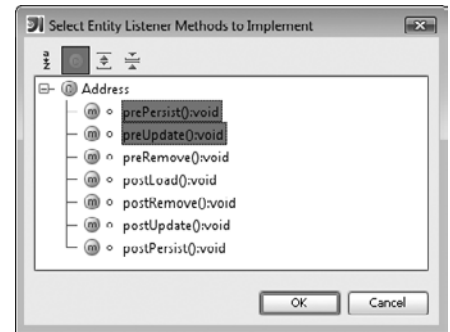


Figure E-10
Sélection des méthodes de callback

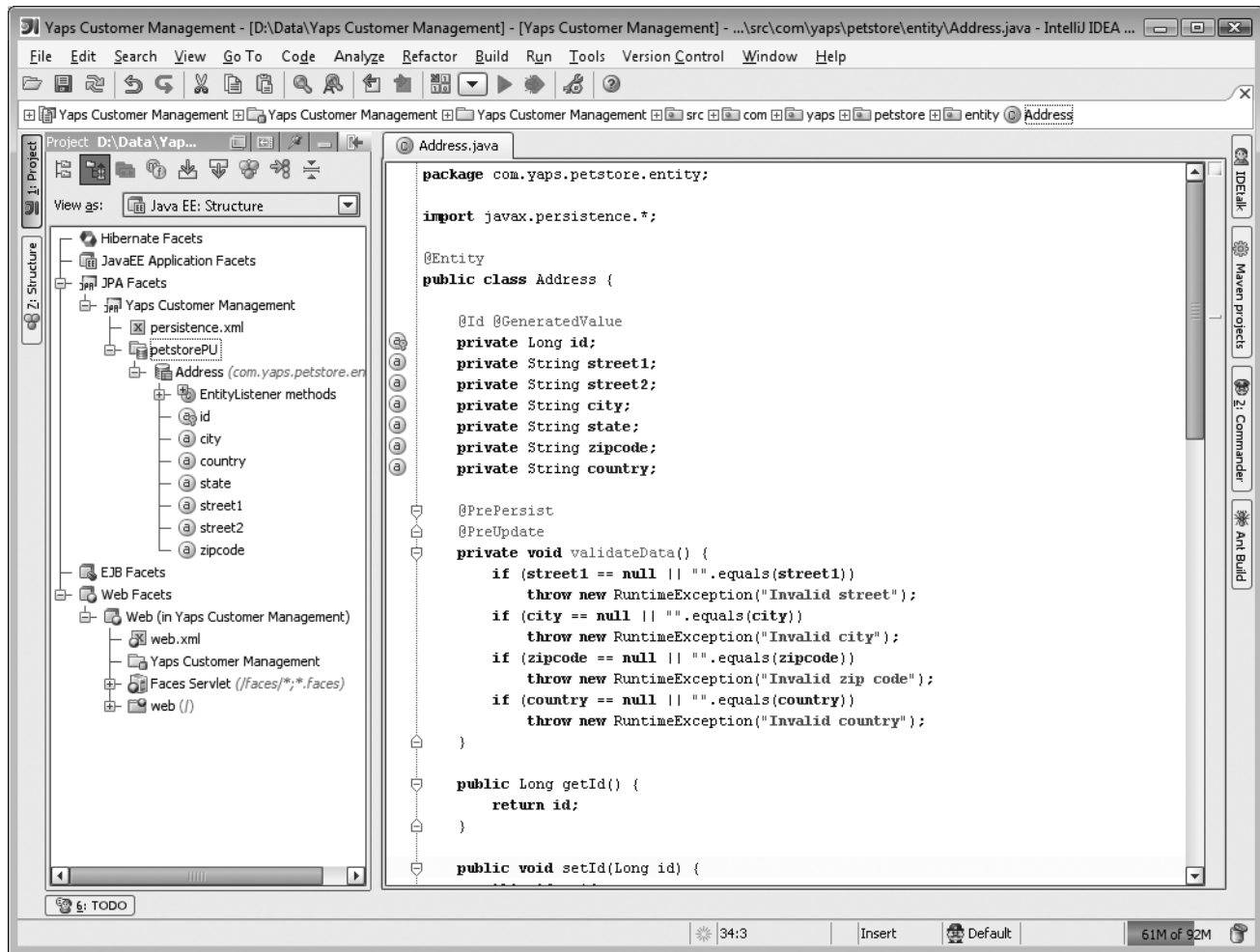


Figure E-11 Entity Address avec attributs et méthode de validation

Répétez ces mêmes actions pour créer l'entité *Customer* (client) avec les attributs *id*, *login*, *password*, *firstname*, *lastname*, *telephone*, *email*, *dateOfBirth* et *age*. Pour créer la relation entre le client et son adresse de domiciliation, nous allons utiliser un diagramme d'entités-relations.

Diagramme d'entités-relations de JPA

Après avoir créé les entités *Customer* et *Address*, nous devons spécifier leurs relations. Pour cela, nous pouvons utiliser le diagramme d'entités-relations d'IntelliJ IDEA. Il est en effet plus facile de faire des liens de manière visuelle.

Faites un clic droit sur l'unité de persistance, puis choisissez le diagramme d'entités-relations (*Open ER Diagram*). IntelliJ IDEA affiche alors les entités définies dans l'unité de persistance (figure E-12).

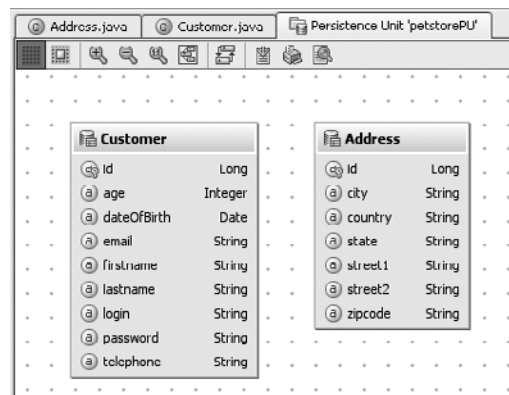


Figure E-12
Diagramme d'entités-relations

Pour créer un lien unidirectionnel entre les deux entités, cliquez sur *Customer* et déplacez la souris vers *Address*. IntelliJ IDEA affiche une boîte de dialogue vous laissant spécifier les propriétés de la relation. Saisissez le nom du nouvel attribut (*homeAddress*) qui sera créé dans la classe *Customer*, puis cliquez sur *OK* (figure E-13).

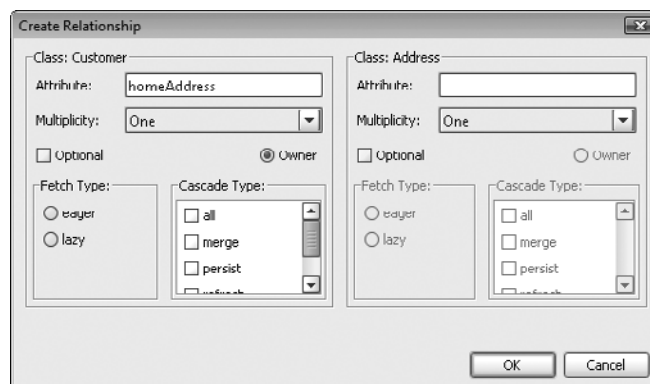


Figure E-13
Propriété de la relation
entre *Customer* et *Address*

Comme vous pouvez le constater, IntelliJ IDEA a créé et annoté le nouvel attribut.

Créer l'interface JSF

Maintenant que le modèle de données est prêt, nous devons créer l'interface web permettant aux utilisateurs de créer, afficher et modifier les données de leurs clients. Pour cela, nous utiliserons le générateur de pages JSF.

Faites un clic droit sur l'unité de persistance `petstorePU` et sélectionnez le menu *Generate Faces Pages*. Cette option n'est disponible que quand la facette JSF est correctement configurée et ajoutée au module, ce que nous avons fait lors de la création du projet. IntelliJ IDEA affiche alors une boîte de dialogue vous laissant choisir le code que vous voulez générer (via des templates totalement ajustables) (figure E-14).

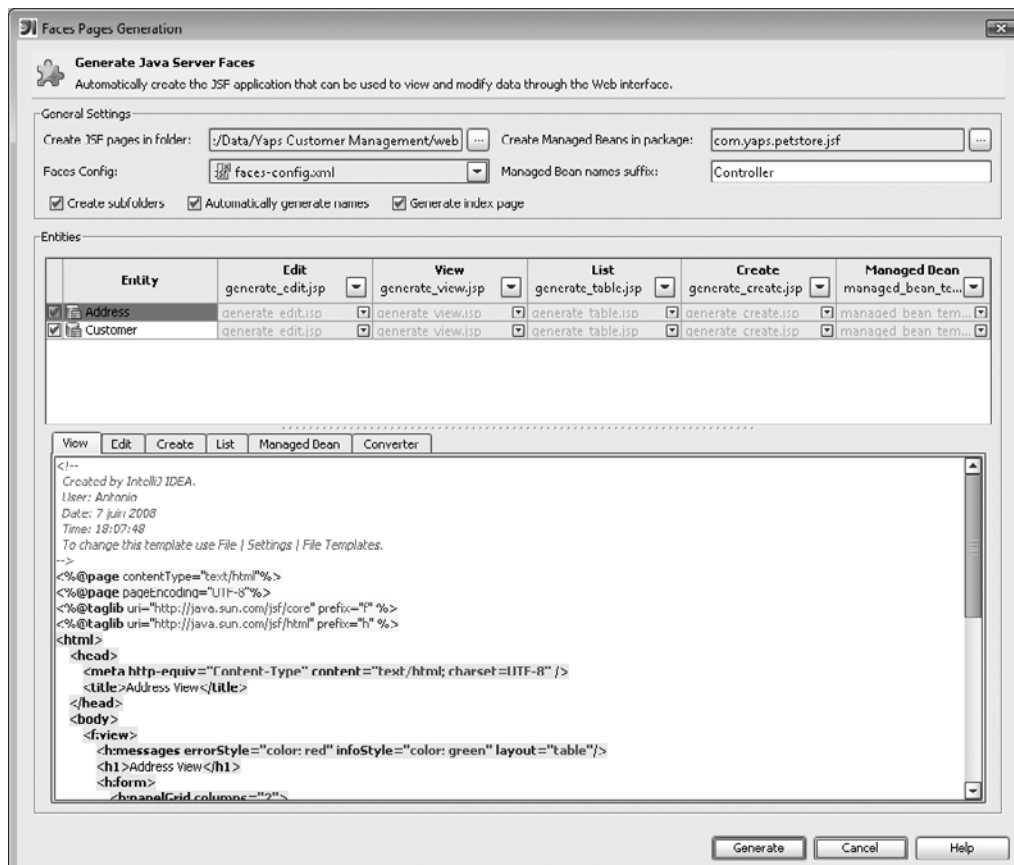


Figure E-14
Génération des pages
pour client et adresse

Sélectionnez le dossier où vous voulez placer les pages JSP générées (répertoire web), le paquetage des managed beans (com.yaps.petstore.jsf) ainsi que le suffixe utilisé pour les managed beans (Controller). Sélectionnez le nom du fichier de configuration JSF (faces-config.xml) qui contiendra les informations sur la structure (les validateurs, les beans, les règles de navigation, etc.).

La table *Entities* affiche la liste des entités ainsi que les pages JSP dont vous avez besoin (pour créer, éditer, consulter, lister). Sous cette table, vous pouvez voir l'éditeur de code montrant les éléments qui vont être générés. Entièrement personnalisable, cet éditeur vous permet de modifier le code comme vous le souhaitez. Cliquez sur *Generate*.

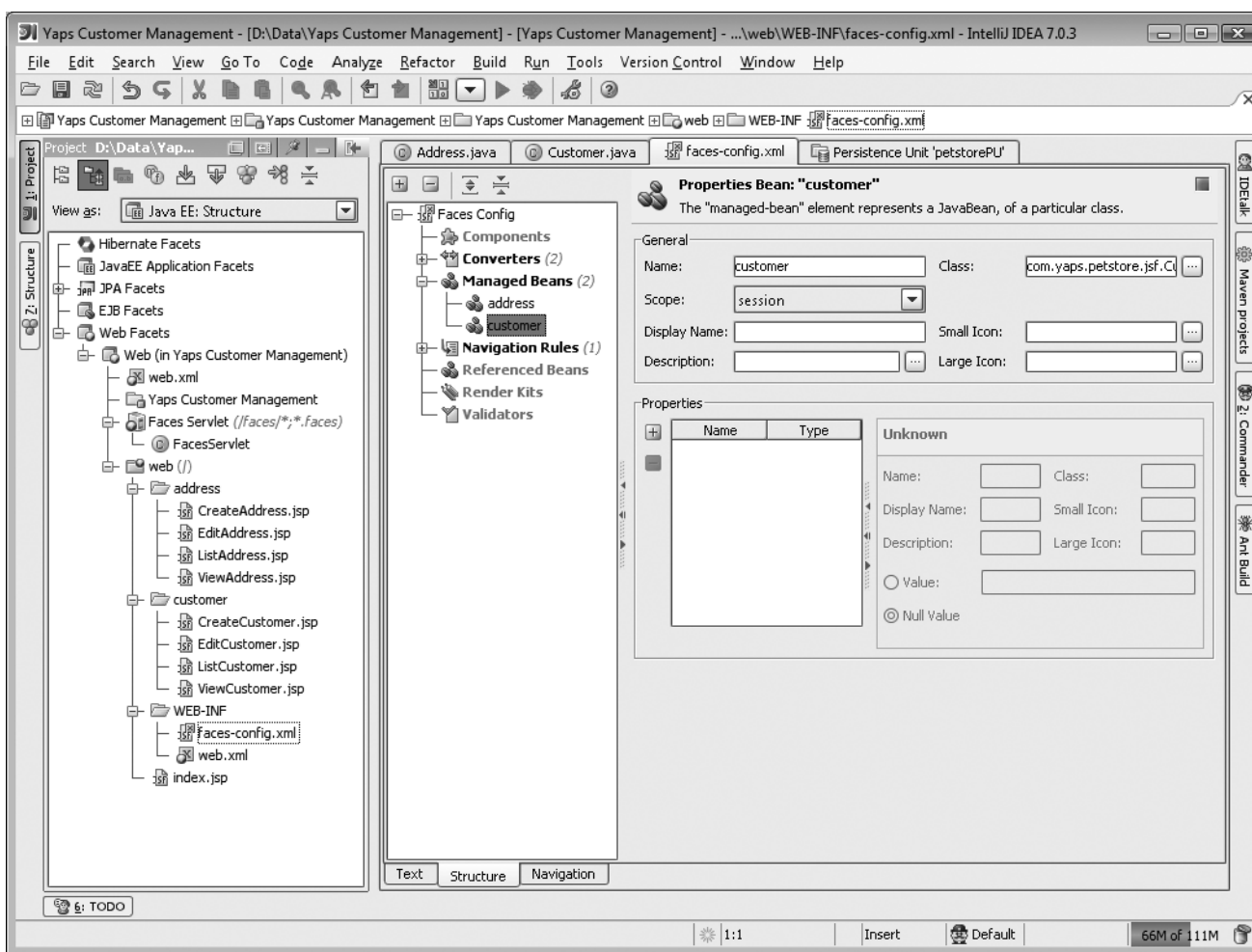


Figure E-15 Édition du fichier faces-config.xml

Les pages JSP sont alors créées, ainsi que les managed beans. Double-cliquez sur le fichier `faces-config.xml` pour ouvrir l'éditeur (figure E-15). Vous pouvez alors modifier les convertisseurs, les managed beans, les règles de navigation et tous les autres éléments de configuration JSF. L'onglet de navigation vous montre le diagramme de navigation. Grâce à ce diagramme, vous pouvez directement passer des éléments visuels (page, navigation...) au code (figure E-16).

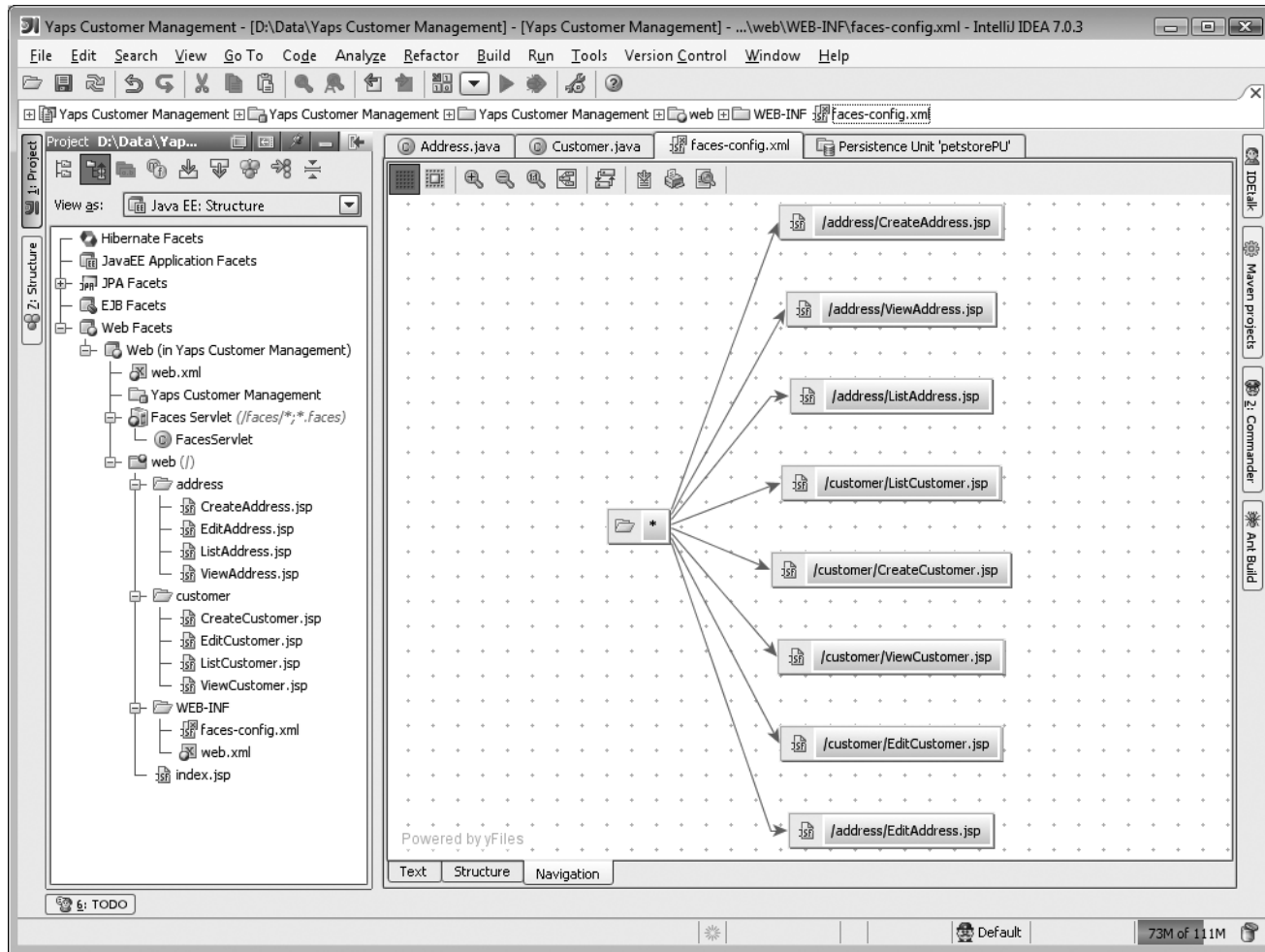


Figure E-16 Navigation entre les pages JSP

Déployer et exécuter l'application

Le code enfin écrit, il faut maintenant le déployer dans GlassFish pour utiliser l'application. IntelliJ IDEA intègre nativement une très large variété de serveurs d'applications, vous permettant de facilement déployer, exécuter et déboguer les applications Java EE.

Les serveurs d'application sont configurés via le menu *Run > Edit Configurations*. Là, vous trouverez toute la configuration nécessaire pour déployer, exécuter et déboguer l'application (figure E-17). Cliquez sur le bouton **+** puis choisissez *Glassfish* et *Local*. Vous pouvez aussi choisir *Remote* si le serveur sur lequel vous voulez déployer tourne sur une machine distante.

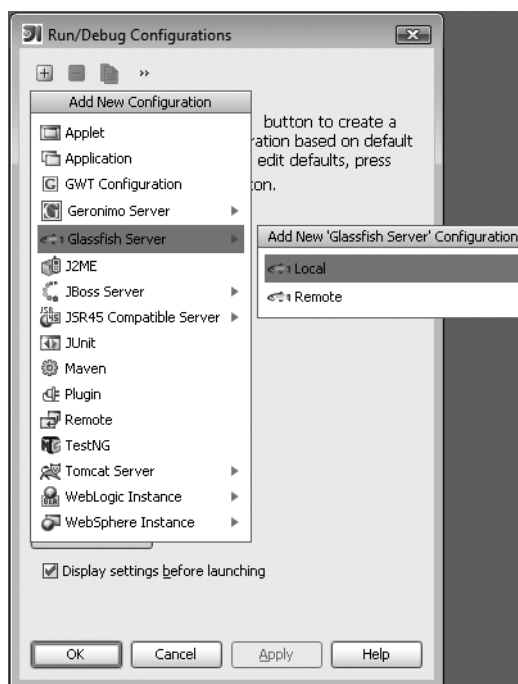


Figure E-17
Sélection du serveur GlassFish

Une fenêtre de configuration s'affiche (figure E-18). Tout d'abord, saisissez le nom de la configuration, par exemple, *GlassfishConfiguration*. Dans l'onglet *Server*, cliquez sur *Configure* et spécifiez le répertoire d'installation de GlassFish. IntelliJ IDEA détecte alors la version du serveur. Sélectionnez la facette JSF à déployer ainsi que le domaine par défaut de GlassFish (*domain1*). Pour ce qui est de la page d'accueil, choisissez *http://localhost:8080/yaps*.

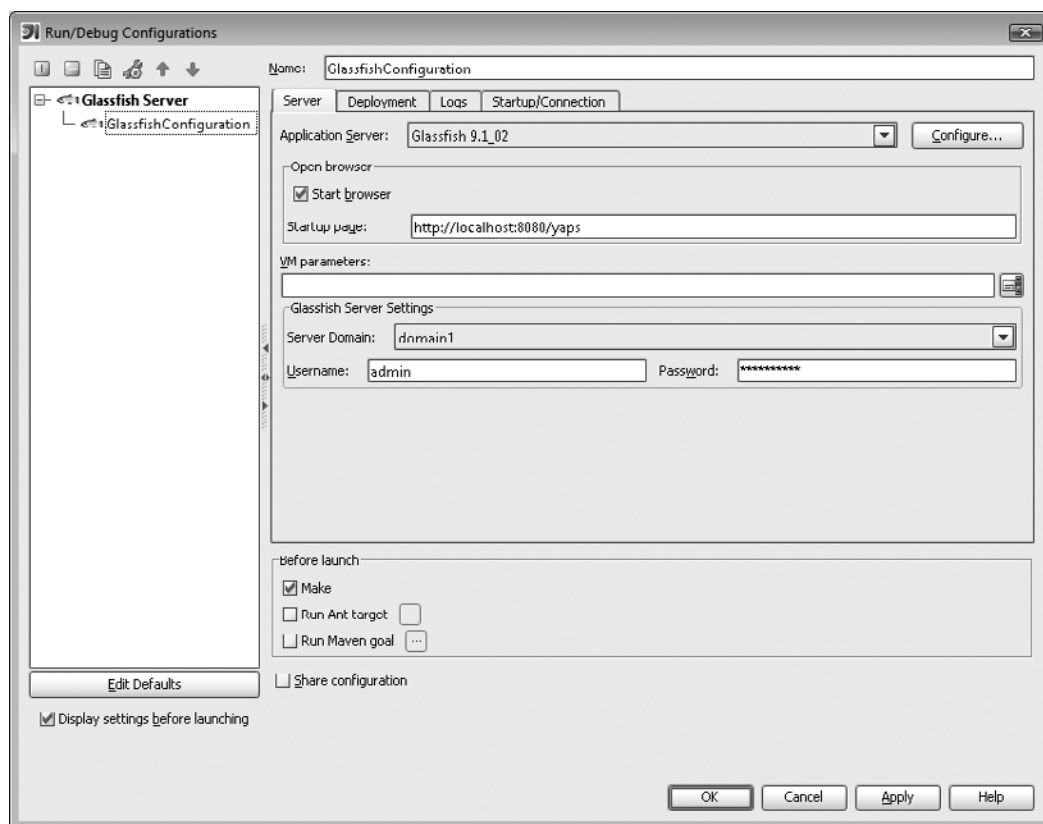


Figure E-18 Configuration du serveur GlassFish

Cliquez sur *OK* pour sauvegarder la configuration. Il ne vous reste plus qu'à démarrer GlassFish et à déployer l'application. Pour cela, vous pouvez appuyer sur les touches *Shift+F10*. IntelliJ IDEA réaffiche la configuration du serveur avant d'exécuter l'application, de sorte que vous puissiez apporter des modifications.

Cliquez ensuite sur *Run*. IntelliJ IDEA compile l'application, démarre GlassFish, crée les fichiers de déploiement, les déploie puis exécute l'application en démarrant un navigateur. Toutes les traces de ces opérations apparaissent alors dans la console d'IntelliJ IDEA, ce qui facilite le monitoring de l'application (figure E-19).

Il ne vous reste plus qu'à vous rendre à l'adresse `http://localhost:8080/yaps/index.faces` pour pouvoir utiliser l'application et ainsi créer, modifier et lister les clients et adresses du système (figure E-20).

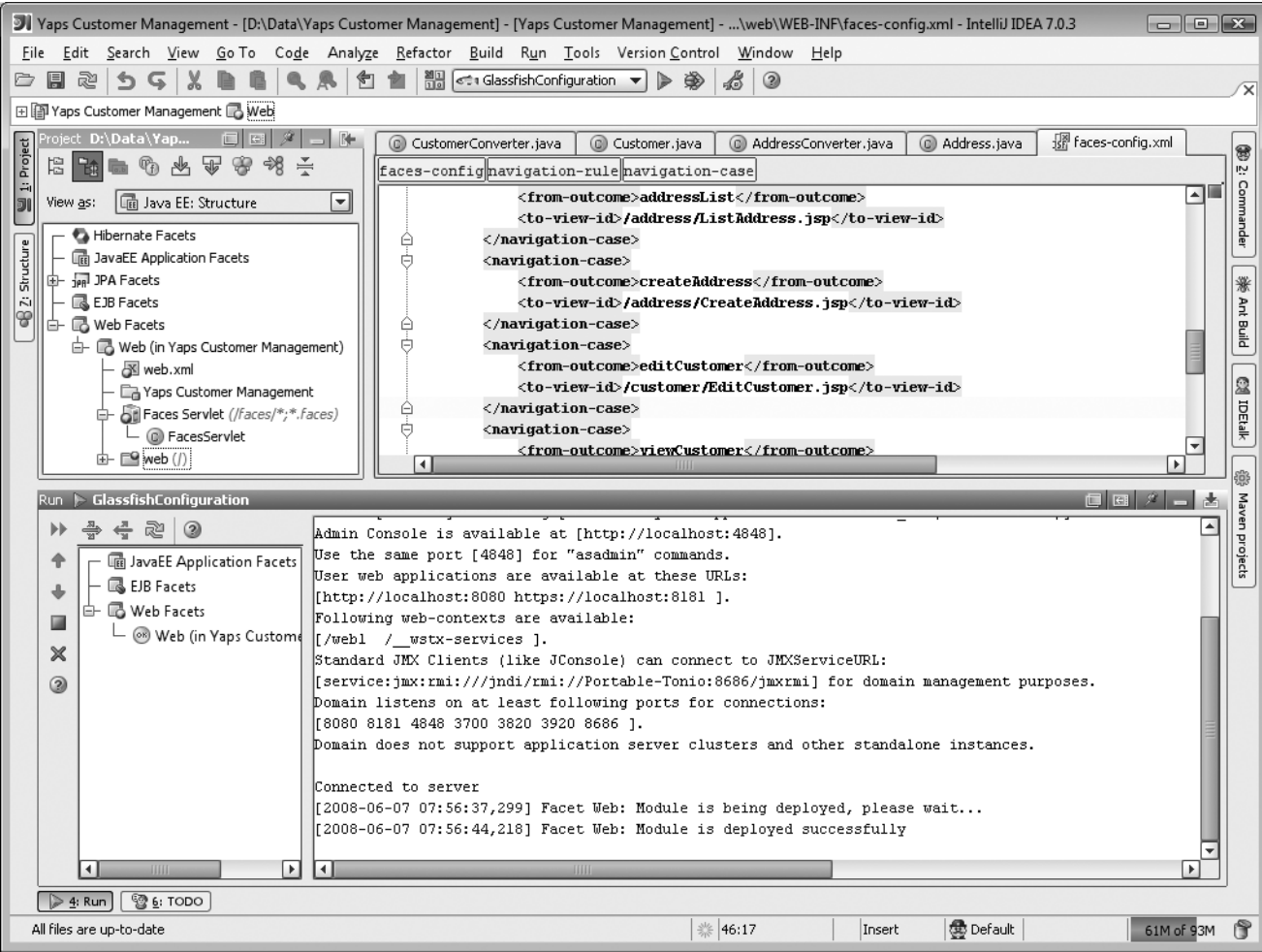


Figure E-19 Traces du serveur GlassFish dans IntelliJ IDEA



Figure E-20
Page d'accueil de l'application générée

En résumé

Les fonctionnalités d'IntelliJ IDEA décrites ici concernent le support de Java EE 5. En effet, un IDE intégré et robuste peut augmenter la productivité des développements. Mais il y a encore beaucoup plus à explorer dans IntelliJ IDEA, puisqu'on observe la même simplicité d'utilisation pour les EJB, les services web, Ajax, etc.

IntelliJ IDEA est sans doute à l'heure qu'il est la solution la plus aboutie en matière d'assistance aux projets de développement. En effet, au vu de la taille et de la complexité que peuvent atteindre ces projets aujourd'hui, on ne saurait négliger l'importance d'un bon outil d'accompagnement alliant puissance, flexibilité et intelligence pour une productivité optimale.

Index

Symboles

@ 72, 87

A

ACC (Application Client
Container) 145, 183

acronymes 27, 317

annotation 26, 27, 35, 72

 @javax.annotation.PostConstruct 129,
 222

 @javax.annotation.PreDestroy 129,
 222, 281, 285

 @javax.annotation.Resource 274, 276

 @javax.ejb.ActivationConfigProperty
 279

 @javax.ejb.ApplicationException 128

 @javax.ejb.EJB 227, 271

 @javax.ejb.Local 110

 @javax.ejb.MessageDriven 35

 @javax.ejb.PostActivate 223

 @javax.ejb.PrePassivate 223

 @javax.ejb.Remote 110

 @javax.ejb.Stateful 34, 224

 @javax.ejb.Stateless 34, 110, 157

 @javax.ejb.TransactionAttribute 125,
 132

 @javax.jws.Oneway 251, 256

 @javax.jws.WebMethod 255, 256

 @javax.jws.WebParam 255

 @javax.jws.WebResult 255

 @javax.jws.WebService 255, 256

 @javax.persistence.Column 74, 96,
 98, 103

 @javax.persistence.Embeddable 78,
 103

 @javax.persistence.Embedded 78, 101

 @javax.persistence.Entity 35, 70, 95,
 98, 101

 @javax.persistence.GeneratedValue 73,
 96, 101

 @javax.persistence.Id 70, 73, 96, 101

 @javax.persistence.JoinColumn 80,
 96, 98, 101

 @javax.persistence.JoinTable 85, 101

 @javax.persistence.ManyToOne 83,
 101

 @javax.persistence.OneToMany 83,
 95, 96, 101

 @javax.persistence.OneToOne 80, 98

 @javax.persistence.OrderBy 95

 @javax.persistence.PersistenceContext
 109, 115, 116, 132, 135

 @javax.persistence.PostLoad 92, 99

 @javax.persistence.PostPersist 92, 99

 @javax.persistence.PostRemove 92

 @javax.persistence.PostUpdate 92, 99

 @javax.persistence.PrePersist 92, 95,
 102, 210

 @javax.persistence.PreRemove 92

 @javax.persistence.PreUpdate 92, 95

 @javax.persistence.Table 71, 96, 101

 @javax.persistence.Temporal 76, 98,
 101

 @javax.persistence.Transient 77, 98

 @javax.xml.ws.WebServiceRef 258
 callback 92, 129, 222, 281

Ant 48, 52, 55

 admin.xml 52, 57, 64, 309

 build.xml 52, 64, 297

 tâche 57, 58, 59, 157, 158

 barkbank 262

 barkbank-build 261, 304

 barkbank-clean 261, 300

 barkbank-compile 261, 301

 barkbank-deploy 262, 306

 create-connection-pool 311

 create-datasource 312

 create-jms-connection-
 factory 60, 314

 create-jms-topic 60, 314

 db-insert-data 306, 309

 list-connection-pool 59, 312

 list-datasource 59, 312

 list-jms-resources 60, 314

 petex 262

 petex-build 261, 304

 petex-clean 261, 300

 petex-compile 261, 301

 petex-deploy 262, 307

 ping-connection-pool 59, 312

 run-client 159, 308

 set-loggers 61, 315

 setup 58, 316

 start-db 311

 start-domain 310

 stop-db 311

 stop-domain 310

 yaps-build 155, 212, 236, 293,
 302

 yaps-clean 155, 236, 293, 300

 yaps-compile 155, 212, 301,
 302

 yaps-deploy 157, 213, 236,
 262, 293, 306

 yaps-undeploy 262

anti-patterns 41

API (Application Programming
Interface) 26

architecture 42, 44

 couche d'interopérabilité 44, 245

 couche de mapping 44, 69

 couche de navigation 43, 180

 couche de persistance 44, 68

 couche de présentation 43, 150

 couche de traitement 44, 108

 couplage lâche 285

- Open Source 31
- séparation des responsabilités 108, 169
- archive
 - barkbank.war 261, 298, 304
 - entity.jar 156, 213, 237, 298, 303
 - mdb.jar 293, 298, 303
 - petex.war 261, 298, 304
 - petstore.ear 157, 213, 236, 262, 293, 298, 304
 - petstore.jar 156, 298, 302
 - petstore.war 213, 237, 298, 304
 - stateful.jar 236, 298, 303
 - stateless.jar 156, 213, 236, 298, 302
 - utility.jar 156, 213, 298, 303
 - ws-interface.jar 262, 298, 303
- ASP (Active Server Page) 164
- asynchrone 32, 35, 43, 60, 268
- autoboxing 26
- B**
 - B2B 240
 - BarkBank 2, 4, 44, 64, 130, 240, 255, 260
 - BLOB (Binary Large Object) 94
 - Blueprint X, 25, 39
 - BMP (Bean-Managed Persistence) 321
 - BMT (Bean-Managed Transaction) 124
 - Business Delegate
 - CatalogDelegate 147, 148, 152
 - CustomerDelegate 147, 152
 - OrderDelegate 147, 152
- C**
 - cas d'utilisation 2, 4, 16
 - Acheter des articles 17, 22, 223, 240
 - acteur 2, 3
 - Consulter et modifier son compte 16, 164, 199
 - Créer un bon de commande 22, 240
 - exception 4, 6
 - Gérer le catalogue 7, 142
 - Gérer les clients 5, 16, 142
 - post-condition 4, 13
 - pré-condition 4, 14
 - Rechercher un article 11, 164, 192
 - Se connecter et se déconnecter 14, 16, 17, 164, 199
 - Se créer un compte 12, 164, 199
 - Visualiser et supprimer les commandes 22, 142
 - Visualiser les articles du catalogue 8, 164, 192
 - CGI (Common Gateway Interface) 164
 - classpath 155, 160, 213, 299, 310
 - CMP (Container-Managed Persistence) 321
 - CMT (Container-Managed Transaction) 124
 - commit 123, 274
 - conteneur 35, 38, 124, 212
 - contexte de persistance 115
 - CRUD (Create, Retrieve/Read, Update, Delete) 44, 108, 135
 - D**
 - DAO (Data Access Object) 115
 - DDL (Data Definition Language) 71, 86
 - ADDRESS 71
 - schéma de la base de données 104
 - T_ADDRESS 76
 - T_CATEGORY 84
 - T_CUSTOMER 82
 - T_ORDER 78, 81
 - T_PRODUCT 84
 - Derby 49, 157
 - création de la base de données 59
 - schéma de la base de données 104
 - descripteurs XML 71, 156
 - design pattern 41
 - Business Delegate 147, 152
 - Composite 173
 - DAO 68
 - DTO 112
 - Façade 109
 - MVC 167, 170, 172, 257
 - Observateur 173
 - Proxy 243
 - Service Locator 145, 289
 - singleton 146
 - Value Object 112
 - E**
 - Eclipse 49
 - EJB (Enterprise Java Bean) 33, 296
 - entity 35, 70
 - stateful 34, 218, 222, 237
 - stateless 33, 34, 108, 222
 - EJBQL (Enterprise Java Beans Query Language) 120
 - EL (Expression Language) 170, 173
 - entity 26, 35, 44, 70, 78, 80, 89
 - Address 70, 72, 93, 97, 99, 131, 133, 199, 227, 259, 285
 - Category 83, 93, 94, 127, 135, 143, 145, 192, 286
 - CreditCard 93, 100, 103, 227, 253, 258, 259, 285
 - Customer 77, 93, 97, 98, 131, 132, 199, 210, 227, 259, 285
 - cycle de vie 91
 - Item 93, 94, 96, 135, 192, 226
 - Order 85, 93, 100, 101, 227, 259, 285, 286, 287, 288
 - OrderLine 93, 100, 102, 259, 285, 288
 - Product 83, 93, 94, 95, 135, 192
 - entity beans 2.x 69, 321
 - BMP (Bean-Managed Persistence) 321
 - CMP (Container-Managed Persistence) 321
 - EJBHome 322
 - EJBLocalHome 322
 - EntityBean 323
 - entity manager 109, 114, 115
 - clear() 116, 119
 - createQuery() 116, 133
 - find() 116, 118, 133
 - flush() 116, 120
 - merge() 116, 118, 120, 133, 259
 - persist() 116, 117, 120, 133, 259
 - remove() 116, 120, 133
 - exception 126, 209
 - application 126
 - CreditCardException 127, 137, 258
 - RemoteException 111
 - système 128
 - ValidationException 128, 133, 137, 152, 210

extension

- .css 164, 190
- .ear 63, 156, 213
- .jar 156, 213
- .jsp 166, 178
- .jspx 166
- .jspx 166
- .war 156, 213

F

- faces-config.xml 181, 184, 213
- FacesServlet 181, 189
- FOP (Formatting Objects Processor) 289

G

- génériques 26, 28, 83, 84
- GlassFish 48, 52, 55, 157
 - asadmin 56, 157, 299, 309
 - configurer le pool de MDB 280
 - création d'un domaine 55, 56
 - création d'un pool de connexion 58, 311
 - création d'une source de données 59, 312
 - création de loggers 61, 315
 - création des ressources JMS 60
 - JNDI 159, 237
 - logs 61, 157, 263
 - mots de passe 56
 - paramètres JNDI 144
 - pool de connexions 58
- GoF (Gang of Four) 41

H

- Hibernate 32, 69, 158, 226
- HTML (HyperText Markup Language) 30, 36, 43, 164, 166, 178
- HTTP (HyperText Transfer Protocol) 36, 164, 168, 188
- HttpSession 188, 200, 218

I

- injection 183, 271
- IntelliJ IDEA XII, 49, 50

J

- J2EE 29, 32, 321
- J2SE 32
- JAAS (Java Authentication and Authorization Service) 203
- Java 26
 - annotation 27
 - autoboxing 26
 - checked exceptions 126
 - collections 83
 - constructeur par défaut 70, 92
 - EE (Enterprise Edition) IX, 31, 49, 295, 321
 - générique 28
 - impression 289
 - passage par valeur et référence 110
 - Pet Store X, 40
 - SE (Standard Edition) 26, 32
 - Threads 268
 - transient 77
 - types énumérés 28
 - unchecked exceptions 126
 - Web Start 157
- Java EE *voir* Java
- Java SE *voir* Java
- JavaMail 38, 44, 281, 296
 - InternetAddress 283
 - message 282
 - MIME 282
 - session 282
 - SMTP 282
 - transport 283
- JAXB (Java Architecture for XML Binding) 38, 243, 289, 295
- JAX-RPC (Java API for XML-based Remote Procedure Call) 243, 295
- JAX-WS (Java API for XML-based RPC) 44, 242, 295
- JBoss 321
- JBoss Seam 190
- JCP (Java Community Process) 31
- JDBC (Java Data Base Connectivity) 29, 44, 68
 - Pet Store 25
 - pilotes 29
- JDK (Java Development Kit) 48, 50, 55

- JDO (Java Data Object) 32, 69

- JEE *voir* Java EE

- JMS (Java Message Service) IX, 32, 60, 268, 296

- acquittement 276
- destination 272
- fabrique de connexions 271
- interopérabilité 280
- message 269
 - corps 270
 - en-tête 269
 - propriétés 270, 286
- MessageListener 275, 279
- objets administrés 271
- point à point 273
- publication/abonnement 273
- sélection de messages 277, 289
- JNDI (Java Naming and Directory Interface) 29, 143
 - ejb/stateless/Catalog 135, 143, 144, 159
 - ejb/stateless/Customer 114, 132, 148, 159
 - ejb/stateless/Order 130, 159
- jdbc/petstoreDS 59, 63, 115, 116, 158
- jms/petstoreConnectionFactory 60, 63, 271, 285, 290
- jms/topic/order 60, 63, 271, 284, 285, 288, 290
- jndi.properties 144
- PortableRemoteObject 144
- JPA (Java Persistence API) 32, 69, 88, 296
 - cascade 89, 117
 - chargement d'une association 87
 - contexte de persistance 115
 - entity manager 109, 114
 - FetchType.EAGER 96, 97, 98, 101, 102
 - FetchType.LAZY 95, 96
 - générer la base de données 76
 - langage de requête 120
 - mettre à jour un entity 119
 - NativeQuery 121
 - ordonner une association multiple 88

- persister un entity 117
- programmation par exception 80
- rattacher un entity 118
- rechercher un entity par son
 - identifiant 118
- relation bidirectionnelle 1:n 82
- relation unidirectionnelle
 - 0:1 82
 - 1:1 80
 - 1:n 85
- supprimer un entity 120
- JPQL (Java Persistence Query Language) 120, 121, 122, 123
 - createQuery() 121
 - getResultList() 122, 134, 135
 - getSingleResult() 121
 - jokers 123
 - Query 121
 - setParameter() 121, 135
- JSE *voir* Java SE
- JSF (Java Server Faces) 38, 43, 172, 295
 - balises Core 176
 - balises HTML 174
 - contexte 194
 - Convertir 176
 - cycle de vie d'une page 179, 181
 - faces-config.xml 181, 184, 213
 - FacesContext 210
 - FacesServlet 181, 189, 213
 - gestion événementielle 180
 - navigation 180, 184
 - clé de navigation à null 186
 - dynamique 186
 - statique 185
 - UIComponent 176
- JSP (Java Server Pages) 36, 43, 164, 166, 172, 295
 - balises 167
 - déclaration 167
 - directive 167
 - expression 167
 - scriptlet 167, 169, 170
- JSR (Java Specification Requests) 32, 295
- JSTL (JSP Standard Tag Library) 37, 43, 170, 173, 295
 - custom tags 171
- JVM (Java Virtual Machine) 68, 91
- L**
- langage d'expression 37, 170, 173
 - unifié 173, 179
- logging 62, 136
- M**
- managed bean 182, 258
 - AccountController 185, 186, 199, 211, 234
 - CatalogController 182, 188, 192, 211
 - ShoppingCartController 226, 236, 253, 257, 260
- MANIFEST.MF 156
- MDB (Message-Driven Bean) 34, 44, 278
 - cycle de vie 280
 - OrderPrinterBean 278
- MOM (Message-Oriented Middleware) 32, 268
- MySQL 321
- N**
- NetBeans 49
- O**
- OMG (Object Management Group) 41
- Open Source 48, 172, 190, 289
- P**
- page JSP
 - confirmorder.jsp 228, 257
 - createaccount.jsp 200
 - footer.jspf 190, 191
 - header.jspf 190, 191, 201
 - navigation.jspf 190, 191
 - orderconfirmed.jsp 228
 - showaccount.jsp 200
 - showcart.jsp 228
 - showitem.jsp 228
 - showitems.jsp 194, 228
 - showproducts.jsp 194
 - signon.jsp 200
 - updateaccount.jsp 200
- paquetages 103, 137, 211, 226, 260, 292
- pare-feu 283
- persistence.xml 116, 156
- PetEx 2, 4, 44, 64, 130, 240, 256, 260
- petstore.css 211
- petstoreDB 59, 63
- petstorePool 59, 63
- petstorePU 116, 158
- PHP (Hypertext Preprocessor) 164
- Pojo (Plain Old Java Object) 35, 70, 113, 236, 326
- protocoles de messagerie 281
- R**
- RMI (Remote Method Invocation) 43
- rollback 123, 124, 127, 273
- S**
- sérialisation 68, 77, 95
- serveur d'application 33, 212
- service web 39, 240, 245
 - artefacts 251, 252, 305
 - classe 250
 - Delivery 248
 - tester 262
 - ValidateCreditCard 252
 - ValidateCreditCardResponse 252
 - validation 246
 - wsgen 299
 - wsimport 299
- servlet 36, 164, 172, 296
 - j_security_check 203
 - scope d'un objet 169
 - ServletRequest 168
 - ServletResponse 168
- session http 188, 200, 218
- SGML (Standard Generalized Markup Language) 30
- SOA (Service-Oriented Architecture) 245
- Soap (Simple Object Access Protocol) 39, 240, 263, 295
- source de données 59
- SQL (Standard Query Language) 44, 68, 120
- SSO (Single Sign On) 203
- stateful bean 34, 218, 222, 237
 - activation 222
 - CartItem 226, 236, 259, 285
 - classe 221

- cycle de vie 222
- interfaces 221
- passivation 222
- ShoppingCartBean 219, 224, 236
- ShoppingCartLocal 219, 224
- stateless bean 33, 44, 108, 130, 222, 259
 - CatalogBean 130, 135, 143, 192
 - CatalogLocal 135, 189, 192
 - CatalogRemote 143, 147, 156
- classe 113
- CustomerBean 109, 113, 115, 130, 132, 199
- CustomerLocal 131, 200
- CustomerRemote 109, 132, 147, 156
- cycle de vie 129
- interface distante 111
- interface locale 112, 148
- OrderBean 130, 258, 260, 275, 285, 292
- OrderLocal 130, 227
- OrderRemote 130, 147, 156
- stateless bean 2.x 111, 326
 - EJBObject 326
 - SessionBean 327
- stéréotype 100, 137
 - <<boundary>> 257
 - <<component>> 137
 - <<control>> 257
 - <<device>> 45
 - <<entity>> 100, 137, 257
 - <<executionEnvironment>> 45
 - <<Extend>> 3
 - <<Include>> 3
 - <<interface>> 137
 - <<Session Bean>> 137
- Struts 38, 172
- Swing 29, 142
 - Look & Feel 150
- synchrone 34, 268
- T**
- Thread 268
- TopLink 32, 49, 69, 158
- transaction 124, 125
 - ACIDité 124
 - explicite 124
 - JMS 274
- U**
- UDDI (Universal Description, Discovery and Integration) 241
- UEL (Unified Expression Language) 173, 179
- UML (Unified Modeling Language) 41, 50
 - attribut dérivé 97
 - classes 131
 - composants 137
 - créateurs du langage 2
 - diagramme d'activités 9
 - diagramme d'états 87, 89, 91
 - diagramme de cas d'utilisation 3
 - diagramme de classes 94
 - diagramme de déploiement 45
 - diagramme de paquetages 42
 - diagramme de séquences 148
 - différents liens 134
 - paquetage 42
 - sous-système 42
 - stéréotypes 100
 - visibilité 134
- URL (Uniform Resource Locator) 240
- V**
- variable
 - ANT_HOME 52, 54
 - AS_ADMIN_PASSWORD 56
 - classpath 155
 - GLASSFISH_HOME 54, 145
 - JAVA_HOME 51, 54
 - path 52
- Visual Paradigm XII, 50, 93
- W**
- W3C 30, 241
- web.xml 213, 237
- WebServiceRef 246
- WML (Wireless Markup Language) 173, 179
- WSDL (Web Service Description Language) 39, 241, 261
- wsgen 251, 256
- WS-I (Web Services Interoperability) 241
- wsimport 252, 256, 260
- X**
- xDoclet 69
- XHTML (eXtensible HyperText Markup Language) 30, 166
- XML (eXtensible Markup Language) 30, 36, 242, 244, 289
 - DOM (Document Object Model) 245
 - SAX (Simple API for XML) 245
- XML-RPC (Remote Procedure Call) 240
- XSD (XML Schema Definition) 30, 38, 242, 249, 261
- XSL (eXtensible Stylesheet Language) 289
- Y**
- YAPS Pet Store 2, 42, 64, 255, 256

Programmez intelligent avec les Cahiers du Programmeur

Java EE 5

EJB 3.0 • JPA • JSP • JSF • Web Services (JAXB, JAX-WS) • JavaMail • JMS • GlassFish • Ant • Derby

Ce cahier détaille la conception d'un site de e-commerce avec UML et Java Enterprise Edition 5. Inspirée du Java Petstore, l'étude de cas se construit au fil des chapitres en appliquant les spécifications Java EE 5 : EJB 3.0, JPA 1.0, Servlet 2.5, JSP 2.1, JSF 1.2, Web Services 1.2, JAXB 2.0, JAX-WS 2.0, JavaMail 1.4, JMS 1.1. L'application est déployée dans le serveur GlassFish et utilise la base de données Derby.

Cet ouvrage s'adresse aux architectes et développeurs confirmés qui veulent découvrir les nouveautés de Java EE 5 ou migrer leurs applications J2EE 1.4 existantes. Il montre comment s'imbriquent les différentes API de Java EE 5 dans une application internet-intranet.



Téléchargez le code source de l'étude de cas !
www.editions-eyrolles.com

Sommaire

Etude de cas • Une entreprise de vente en ligne • Expression des besoins • Les acteurs du système et les cas d'utilisation • **L'architecture de l'application** • Java SE 5 • Java EE 5 (JPA, JMS, EJB, Servlet, JSP, JSF, JavaMail, Web Services) • XML • UML • Le blueprint Java Petstore • Découpage en couches de l'application • **Installation et configuration des outils** • JDK • Ant • GlassFish • L'utilitaire asadmin • La console d'administration de GlassFish • La base de données Derby • **La persistance des données** • Java Persistence API • Annotations JPA • Mapping • Les objets persistants de l'application • Les relations entre objets • Jointures • Pojos et entités • Schéma de la base de données • **Traitements métiers** • Stateless session bean • Pattern Facade • Entity Manager • Opérations CRUD • Contexte de persistance • Le langage de requête JPQL • Démarcation de transactions • Gestion des exceptions • **L'interface swing** • JNDI • Les design patterns Service Locator et Business Delegate • Application Client Container • Compiler, packager, déployer et exécuter l'application • **L'interface web** • Servlet, JSP et JSTL • JSF • Les balises JSF • Le langage d'expression • Managed beans • L'injection • Navigation entre pages • Le pattern MVC • **Conserver l'état dans l'application web** • Stateful session bean • Le panier électronique • **Les échanges b2b** • SOAP, UDDI, WSDL, JAX-WS et JAXB • Messages XML • Les services web • Annotations JAX-WS • Génération et utilisation des artefacts • **Les traitements asynchrones** • JMS • Les files d'attente • Message-driven bean • JavaMail • **Annexes** • Spécifications Java EE 5 • Tâches Ants • EJB 2 • Développement avec IntelliJ IDEA.

Architecte senior, **Antonio Goncalves** intervient comme consultant indépendant sur les technologies Java. Ancien consultant Weblogic chez BEA Systems, il s'est spécialisé depuis 1998 dans l'architecture logicielle et serveurs d'application. Membre de l'OSS Get Together Paris, il préconise à ses clients les outils Open Source. Antonio Goncalves fait partie de l'expert groupe JCP sur les spécifications Java EE 6, EJB 3.1 et JPA 2.0. Il enseigne également au Conservatoire National des Arts et Métiers et rédige des articles techniques pour DevX, developpez.com et Programmez.